jamk.fi

Health Check API and Service Discovery in Microservice Environment

Juuso Minkkilä

Bachelor's thesis
April 2019
Technology, Communication, and Transport
Degree Programme in Software Engineering

jamk.fi

Description

Author(s)	Type of publication	Date
Minkkilä, Juuso	Bachelor's thesis	April 2019
		Language of publication: English
	Number of pages 78	Permission for web publication: Yes

Title of publication

Health Check API and Service Discovery in Microservice Environment

Degree programme

Information and Communication Technology, Software Engineering

Supervisor(s)

Rintamäki, Marko; Huotari, Jouni

Assigned by

Landis+Gyr Oy

Abstract

The thesis was assigned by Landis+Gyr Oy as a part of an initiative to improve the company's microservices. The purpose of the study was to research service discovery system and service health check patterns as ways to ensure the connectivity of healthy microservice instances with a reduced need for configuration. Thus, the objectives set by the company were to research how both patterns could be applied to microservices, what problems they solve, and how the patterns affect microservices. After researching the subjects at hand, a plan was made to create a proof of concept demonstrating them. This proof of concept system would fulfil the most rudimentary requirements for both patterns. During the planning many viable tools such as Consul were found and further inspected. However, only some of those tools were later used during the implementation.

The end goal of the thesis was to create a proof of concept system implementing both service discovery and service specific health check APIs. Two microservices were created with Scala programming language and its Akka-libraries. The services were configured to use Consul as their primary tool for service discovery, allowing them to discover each other properly. Verizon's Helm library was used in the Scala services to interface with Consul's HTTP API. Everpeace's Healthchecks library was used to add health checks for the services along with HTTP endpoints to display services' health.

The primary goals of the thesis were achieved. However, many of the initial side goals such as containerizing the services were left out of the thesis as its scope was narrowed. While the way Consul was used in the thesis is suboptimal for production, the services could discover healthy instances of each other based on the service name alone. This successfully demonstrates the benefits of service discovery and health checks.

Keywords/tags (subjects)

Consul, Service Discovery, Health Check API, Microservices

Miscellaneous (Confidential information)

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Minkkilä, Juuso	Opinnäytetyö, AMK	Huhtikuu 2019
		Julkaisun kieli Englanti
	Sivumäärä 78	Verkkojulkaisulupa myönnetty: Kyllä

Työn nimi

Palveluiden löydettävyys ja terveyden tarkkailu mikropalveluympäristössä

Tutkinto-ohjelma

Ohjelmistotekniikan tutkinto-ohjelma

Työn ohjaaja(t)

Marko Rintamäki, Jouni Huotari

Toimeksiantaja(t)

Landis+Gyr Oy

Tiivistelmä

Opinnäytetyön toimeksiantona oli tutkia palveluiden löytämistä ja niiden terveydentilan tarkkailua helpottavia ohjelmistokehitysmalleja mikropalveluympäristössä. Mallien tarkoituksena on varmistaa, että mikropalvelut löytävät riippuvuutensa mahdollisimman pienellä vaivalla ja kunnossa, jossa ne kykenevät mikropalvelua palvelemaan.

Opinnäytetyön tavoitteina oli tutkia, kuinka tutkittavat mallit – *service discovery* eli palveluiden löydettävyys sekä *health checks* eli terveystarkastukset – vaikuttavat mikropalveluiden toimintaan. Lisäksi tutkittiin, mitä ongelmia kyseiset mallit ratkaisevat, ja kuinka kyseiset mallit voitaisiin ottaa käyttöön niiden toiminnan havainnollistamiseksi.

Aihealueen tutkimisen jälkeen lopullisena päämääränä oli tuottaa esimerkkitoteutus järjestelmästä, joka hyödyntää tutkituista ohjelmistokehitysmalleista molempia. Toteuttamiseen käytettiin aiheiden tutkimisen aikana löydettyjä työkaluja ennalta määriteltyjen teknologioiden lisäksi. Järjestelmää varten kehitettiin kaksi mikropalvelua hyödyntäen Scala-ohjelmointikieltä sekä sen Akka-kirjastoja. Toistensa löytämiseksi mikropalvelut hyödyntävät Consul-palveluverkosto-ohjelmistoa, jonka kanssa kommunikoinnissa hyödynnettiin Verizonin Helm-kirjastoa. Lisäksi palveluille toteutettiin HTTP-rajapinnat, josta voitiin tarkkailla niiden senhetkistä terveydentilaa.

Opinnäytetyön aikana saavutettiin sille asetetut ensisijaiset tavoitteet. Moni opinnäytetyölle aluksi asetetuista sivutavoitteista jäi kuitenkin suorittamatta, sillä tutkimusaiheiden syvyyden selkeytyessä kävi selväksi, että työn laajuutta oli pienennettävä. Lopputuloksesta käy kuitenkin ilmi, että oikein sovellettuna käytetyt ohjelmistokehitysmallit helpottavat palveluiden käyttöönottoa.

Avainsanat (asiasanat)

Consul, Service Discovery, Health Check API, Microservices

Muut tiedot

Contents

Figures

Tables

Acronyms

API	Application Programming Interface
DNS	Domain Name System
JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
OOP	Object Oriented Programming
TTL	Time To Live
VM	Virtual Machine

1 Introduction

Recently, small and easy-to-scale microservices have increased in popularity. While microservices have some benefits over the monolithic services of yesteryears, they have problems of their own. Since microservices often depend on each other, it becomes critical to make sure that they can properly find each other. Additionally, because a service instance that is not working properly is hardly worth connecting to, it is also important to make sure that the different service instances are healthy. While there are several ways to approach these issues, such as service orchestration and service monitoring, this study focused on service discovery and service health check design patterns as ways to alleviate these issues. The purpose of this study was to research both of those subjects, and to create a proof of concept system which demonstrates both in action.

Landis+Gyr Oy is a subsidiary of Landis+Gyr Group, which is an industry leader on the field of energy management solutions (About Landis+Gyr n.d.). With its Finland main office located in Jyskä, Landis+Gyr has offices all around the globe. The company has decided to use microservices for some of their backend components. Since the microservices can depend on each other, it would be optimal if the microservice instances were as easily connectable as possible. Service discovery pattern is one way to solve that issue and seemed worth looking into. It would allow services to find other service instances with just the service's name. Service health checking was also included in the thesis scope as health checks can have great synergy with service discovery. For example, with the correct implementation, a service discovery system could react to changes in services' health and hide unhealthy service instances from clients such as other services or end users entirely. This can reduce the chance that the client connects to an inadequately working service instance. Health checks can also provide easy-to-access insight into the internal state of the service, which can inform the service provider about what is really going on in the service without having to delve into the service's logs.

As the subject matter of the thesis work was foreign, it was decided that the work would start with a general research of both subjects. Since the general theory of the subject matters was one of the focal points of this study, this provided a relatively

wide view to most of the subject matters. Additionally, a note was made of any tools found during the research. As this study also had a heavy emphasis on the practical implementation of service discovery and health checks in a microservice environment, the available toolset for the tasks at hand was quite wide. While the ideas behind the tools were found to be similar within each tool category, only a few select tools were used during the implementation, all of which are researched in later chapters.

2 Defining goals and requirements

2.1 Overall goal

The requirements set in the assignment were quite straightforward. Since microservices that are used in modern large-scale systems can have dependencies to each other, the services need some method to find instances of the services they depend on. During the selection of the study's subjects, service discovery was explicitly mentioned as a design pattern that could help alleviate this issue, and as such it was taken as a primary target for the study. Another possible subject for the study were service health checks, as making sure that service instances are working properly is imperative when operating applications composed from multiple microservices. During the preliminary thesis subject selection phase, it also turned out that health checks can complement certain service discovery systems (such as Consul) quite well by providing the service discovery system with the information indicating whether the service instance is working in a "healthy" way. As such, service health checks were also added as a subject for the study further widening its scope.

The primary requirement and the ultimate goal for this study was to research and create a proof of concept system demonstrating the mechanics and implementation of both service discovery and health checks. In the context of the study this means that the services need to be able to find each other based on the service name alone and should provide an endpoint displaying the current health of the service instance. While several different technologies were researched and used during this study, the technologies themselves are not the focus point of the study. Instead, the focus lies

more on the theory and the ideas behind the implementation. This is to make sure that the patterns can be implemented even if the used technologies or languages need to be changed. Because of this, the design used during the thesis should not rely excessively on concepts and patterns only available in a select few languages or technologies. However, the tools used during the project should be usable in large-scale projects as well and not exclusively in small-scale demonstrations. Additionally, the use of technologies compatible with the ones already used by Landis+Gyr would provide additional value for the thesis. As the project would only serve as a proof of concept, direct correlations for Landis+Gyr's own projects would be kept to a minimum. This should keep the need for redactions and corrections minimal.

2.2 Defining feature requirements

To properly demonstrate both service discovery and health check APIs with a proof of concept, some minimum requirements would need to be defined in order to see that the finished proof of concept does what is was set out to do. To begin with, the purpose of service discovery is to help services to find other services' network locations with minimal information, such as with just the service's name and access to the service discovery system's registry (Xu 2018). Thus, in order to demonstrate service discovery in action, it is required that the system has functionality to enable such actions to take place. Additionally, in order to properly demonstrate service discovery, more than one service is required. Alternatively, one service might be enough if it is designed to work as a mesh or a cluster of multiple instances. However, to keep the design of the services as simple as possible the former would be chosen. For this specific proof of concept this means a simple architecture consisting of a "server" service that provides something, and a "client" service which provides that something.

Some technical requirements were imposed in the assignment itself. For example, to ensure that the principles of the proof of concept system could be mimiced with Landis+Gyr's pre-existing microservices, the services used in the proof of concept would be developed using Scala and its Akka framework.

Since projects can often run into all kinds of issues, it is necessary for the requirements to have some flexibility. As it is likely that not all of the requirements can be feasibly met, the most important ones are prioritized, while the rest are left as stretch goals. The defined functional requirements for the proof of concept can be found in table 1, and technical requirements can be found in table 2. Each of the requirements also has an indicator marking whether they are stretch goals or primary requirements.

Table 1. Functional requirements

ID	Requirement	Priority
FR-01	Services can find each other without prior knowledge of each other's addresses	Primary
FR-02	Services provide data regarding their health	Primary
FR-03	Unhealthy services are not provided to clients via the service discovery system	Primary
FR-04	External services can be registered to the service discovery system	Stretch-goal
FR-05	Service can report health data to service registry using a "Time to Live" health check	Stretch-goal
FR-06	Operations between services are recorded into a database	Stretch-goal

Table 2. Technical requirements

ID	Requirement	Priority
NR-01	"Client" service has a dependency on "Server" service	Primary
NR-02	Services should use Scala with Akka framework	Primary
NR-03	Database contents can be displayed from a service endpoint	Stretch-goal
NR-04	Services run as containers	Stretch-goal
NR-05	Services should also have version made with .NET as well	Stretch-goal

2.3 Scoping

As the study is wrestling with two different yet intertwined subject matters, restricting the study's scope is important. This is to ensure that the study's resources are spent on researching and developing matters most important to the core subjects and to make sure that the study's focus does not drift away from the targeted goals. A well-defined scope should also reduce the risk of so-called *scope creep*, where the project's scope expands during the duration of the project.

2.3.1 Microservice scope

On the front of microservices in general, the scope of the study is quite narrow. It primarily includes general theory on microservices, and anything necessary for building two bare-bones microservices. Since microservice design is not a primary subject of the study, it should be enough to keep the details and knowledge regarding them quite general. The same goes for the microservices created for the proof of concept: advanced Scala and Akka techniques are outside of the reasonable scope of the study, and thus should be kept to an appropriate minimum. Additionally, the testing, deployment, and security of microservices are hardly relevant to the goals of this study, and thus will not be excessively researched or covered either. However, if some of these matters are relevant to a tool covered in this study, those matters are acknowledged within the context of the tool. In addition, while security will not be explicitly covered in this study, decisions actively weakening the security of the proof of concept will be avoided whenever possible. As for deployment, since the proof of concept services must be deployed in some manner to demonstrate their usage, deployment of services certainly must be researched on a surface level. However, advanced deployment concepts such as deployment automation will not be explored.

2.3.2 Service Discovery scope

For the purposes of this study the research on service discovery pattern and the core concepts behind it should be prioritized over the research of tools that can be used to implement a service discovery system. Additionally, if a subject that is tangential to service discovery or derives from it is found during the research, that subject

should be included in the study's scope on a surface level. However, in order to develop a proof of concept system to demonstrate service discovery, the tools for that purpose must be included in the study's scope. To keep things relatively concise, after evaluating available options for service discovery systems, only one of them should be properly included in the study's scope. However, should the tool provide features additional to service discovery, they will not be extensively explored. If other found tools include particularly interesting concepts or features, they should be included in the study's scope on a surface level.

2.3.3 Health Checks scope

As with service discovery, with health checks the study's scope is most concerned with the health check pattern itself. If it is found that there are multiple distinct methods for implementing health checks, each method should be acknowledged and researched to an appropriate level. Extensive research of different health check tools is left out of the scope, though some tools must be found and researched. Additionally, if a subject tangential to health checks is found during the research that will be included into the study's scope whenever appropriate.

3 Theory

3.1 Client-server architecture

The services of the proof of concept system follow a simple client-server architecture. In a client-server architecture, a client requests a resource or a process from the server. The request is processed by the server and the produced result is then delivered to the client to consume. The requests are sent over a network, and thus the architecture is sometimes referred to as network computing model. Additionally, in a client-service architecture the server is capable of handling multiple clients at a time, and the client can connect to several servers at a time. This means that the server and the client are not locked into a restrictive one-to-one connection and are thus able to handle other tasks alongside the connection. (Client/Server Architecture n.d.; What is Client-Server Architecture? n.d.; Client-server architecture n.d.)

3.2 Microservices

According to OASIS, a service is a mechanism that allows a client to access one or more capabilities (Reference Model for Service Oriented Architecture 1.0, 2006, 21). To elaborate, a service could be described as a bundle of various functions that serve to fulfil one or more goals. One or more of these services can be combined into an application. A conventional monolithic application has all its functions crammed into one large executable. Conversely, if a service was split into small logical, loosely coupled units, each of those units could be made into a small independently deployable *microservice*. For example, if there was a user-based chat service, one microservice could handle the login process while another microservice handles the actual chatting itself, as opposed to a monolithic service where both of those processes would be handled by the same service. (Lewis & Fowler 2014; Richardson 2018.)

3.2.1 Microservices versus monoliths

Microservice have some key benefits, many of which stem from their distributed nature and are shared with other distributed systems. Since the application's functions are distributed between different services, one service going down does not necessarily bring down the whole application along with it. This increases the application's error resiliency. Additionally, since the services are separate, they can be developed separately if the boundaries and interfaces between the services have been clearly defined. The separation of different services also means that changes to services can be made independently from the rest of the system. This can increase speed at which pieces of the application (in this case services) can be deployed and released. Clearly defined boundaries and interfaces also enable the usage of different technologies between different services, as long as they provide an interface that fulfils the requirements of their clients. This means that services can use whatever tools are most suitable for their task instead of being bound by the technologies already used by other services in the application. (Newman 2015, 4-8.)

Arguably the biggest advantage of microservices is that they are easier to scale than monoliths. With conventional monoliths, if just one part of the application is running

out of resources and bottle necking the application, a completely new instance of the application must be created. This also includes the service parts that still have plenty of capacity left, as the load on the service may not spread evenly between its different functions. Such a task may often involve a substantial cost if new hardware is required. Conversely, in a system built from microservices, if one microservice is running slowly but the rest are fine, another instance of just that microservice can be deployed. A monolith can also be scaled by running it on a more powerful machine. However, that does not increase the application's resiliency in the same way that having a whole new instance of it does. (Lewis & Fowler 2014; Newman 2015, 217.)

3.2.2 Issues with microservices

The advantages of microservices come with certain drawbacks. For example, communication between services is done using remote calls using the services' public interfaces. Compared to in-process calls remote calls are more expensive. Microservice architecture also has all the complexities associated with distributed systems (Newman 2015, 11). This includes complexities in deployment and management. However, the primary microservice issues investigated during this study all have to do with keeping track of the service instances. As it is often necessary for microservices to request resources from one another, how services find and connect with other services becomes an issue. If services have multiple instances, each one of them must be kept track of. In addition to knowing the location of each service instance it can also be useful to know if the instances are working as expected, capable of serving clients in a healthy way (ibid., 161-162). The primary solutions investigated in this study are the service discovery pattern and service health check pattern. (ibid., 236; Richardson 2018c.)

3.3 Service Discovery Pattern

In a microservice system, it is very likely that not all services run on the same host, behind one IP address and with pre-configured ports. If new instances of a service are deployed, it is possible that its IP is not previously known by the service's clients. This issue is amplified when running the application on a cloud platform using dynamically assigned network locations. As the name implies, the purpose of service

discovery pattern is to make finding services easier. The goal is that every service depending on another service can fulfil those dependencies with as little previously known information as possible. In a nutshell, service discovery pattern allows a service with dependencies ("client") to find other services registered on the same service discovery system. The client requests a service instance from the service discovery system, and depending on the system's design, it either connects the client directly to a service instance selected by the service discovery system; or returns the connection details of available service instances to the client, which can decide which instance to connect to. The former is referred to as "server-side discovery", and the latter as "client-side discovery". Both usually rely on some sort of *service registry* containing the network locations of all the services in the system. (Richardson 2015; Richardson 2018e.)

3.3.1 Client-Side versus Server-Side Service Discovery

The biggest difference between client- and server-side discovery is which party of the discovery chooses the connected service instance. This has a major impact on the load balancing logic of the system. In client-side service discovery, the client requests a list of available service instances from the service discovery. The client then chooses the most suitable instance based on its own logic. Compared to server-side discovery, this is a more nuanced method. Alternatively, with server-side discovery, the logic of the service discovery system or a separate load balancer is used in its stead. This allows the client's service discovery layer to be much lighter; however, some of the nuance in instance selection can be lost when compared to client-side discovery. (Richardson 2015; Richardson 2018d.)

Both methods have their own advantages and disadvantages. With server-side discovery, the details of the service discovery system are hidden away from the client behind a load balancer or some similar component. When a client requests a service instance, the instance selection itself is done *server*-side instead of by the client. Since in server-side discovery the client does not deal with service discovery directly, it can be quite easy to take into use, potentially requiring no additional logic from the client service itself. (Richardson 2015; Richardson 2018d.)

In client-side service discovery, the client itself is responsible for querying the service registry for instances of other services. While having a higher implementation overhead than server-side discovery, it can be advantageous to have service instance selection logic in the client service itself. This way the client can use its own logic for the service instance selection without causing a strain on the service registry server. Additionally, it allows for a great deal more freedom on the choice of service registry. Since the service instance is usually requested explicitly, the required data could be sourced from a multitude of different sources. However, one trade-off is that implementing such logic requires a layer of code for each and every service just for that. Another drawback is that the code required for interacting with the service registry is likely specific to that registry, and as such the service is then coupled to the service registry. (Richardson 2015; Richardson 2018a.)

3.3.2 Service Registry

The core of most service discovery systems is the service registry. How a service registry works, varies from implementation to implementation. However, at its core a service registry is usually some sort of key-value storage. The network locations of the services using the service discovery system are stored in the registry. The network location conventionally consists of the service's IP address and port associated with the service's name. The service's network location is registered either directly by the service itself or a third party. In addition to plain network locations, some service registries such as Consul or etcd can store additional metadata about the service as well, such as various "tags" or service's health check status, either due to the registry's purpose-built features or thanks to their key-value format (Services n.d.; Checks n.d.; Farcic 2015). The service's network location can then be queried from the service discovery system with the service's name, which then fetches the data from the service registry. This behaviour has some parallels to DNS to the degree that DNS systems can be used for service discovery if need be (Newman 2015, 237-238). (Richardson 2015; Richardson 2018e.)

Regardless of the technology used to implement the registry, in large scale applications having it in just one place might become an issue. Since the registry is the source of information for service discovery, having just one instance of it would

cause the system to have a single point of failure. Additionally, depending on the scale of the application and the topology of its network, running only one service registry instance may also cause connectivity and latency issues, either because of network latency or the registry being overloaded with requests. Therefore, in order to increase availability and failure tolerance, it is advisable to always have multiple instances of the same registry available, for example by using a distributed database as the service registry. (Xu 2018; Richardson 2018e.)

3.3.3 Service Mesh

While service mesh solutions were outside of the scope of this study, the theory of service meshes is tangentially related to service discovery. Service mesh solutions as a topic is slightly more advances than service discovery or health checks alone, and it can even fulfil their role to some degree. Service meshes are loosely comparable to server-side service discovery, as in both the service discovery itself is executed outside of the services themselves. The main goal of a service mesh is to have the application's communication layer be a visible member of the application's ecosystem, separate from the services themselves. (Miranda 2016; Morgan 2017.)

In a typical service mesh system, every service instance has a so-called sidecar proxy deployed alongside them. The service's security, monitoring, connection handling, etc. are offloaded to the sidecar proxy, which makes the service itself more lightweight. This way complex logic not directly relevant to the service can be handled elsewhere, and the developers working on the service can focus more on the service instead. Usually the service itself does not have to be aware of the proxy; however, certain solutions and technology combinations also allow services to natively integrate to the service mesh system without the overhead of a proxy sidecar. An example of this is Consul's capability to natively integrate with services created with Go programming language (Connect-Native App Integration). (Miranda 2016; Morgan 2017; Smith 2018.)

3.4 Health checks

Regardless of the type of the service in question, it is useful to know whether it is performing as expected. Some ways to approach the issue are metrics and logs. Both

can give valuable insight to the service's performance but might not tell whether the service is working as expected. Knowing that the service is running does not tell much about the service's capability to handle incoming requests. The same goes for metering and logs; the information that the service is using a certain amount of resources, or that the service's logs have no errors does not automatically mean that the service is working right. Therefore, in addition to logs and metrics, it is useful to make periodic *health checks* on the service to see if it is *healthy* and capable of handling requests. (Ewaschuk 2016; Richardson 2018b.)

The purpose of monitoring service's health is to make sure that the service is in a state where it can handle requests sent to it (Richardson 2018b). Some monitoring systems such as logging and metering rely on data that is not visible from outside of the service and can thus be described as a *white-box* monitoring. On the other hand, *black-box* monitoring targets behaviour that is visible from the outside of the service. Health checks test the service's capability to fulfil requests sent to it by running checks against the service, and as such health checks can be described as black-box monitoring. (Arjis 2018; Ewaschuk 2016.)

White-box and black-box monitoring are not mutually exclusive. In fact, they can often complement each other, so it is highly recommended to use both whenever possible. This way if unhealthy behaviour is noticed, it can then be compared with system logs and metrics to see what caused the drop in the health status. Health checks can also be complementary to service discovery solutions. If a service discovery system is used for the application, health checks can provide additional value, since unhealthy service instances can be marked as such in the service registry. When a client then requests a service instance, the instance marked as unhealthy can be safely ignored ensuring that the client is directed to a service instance capable of handling the client's requests. (Ewaschuk 2016; Richardson 2018b.)

The way that the service registry is informed about changes in service instance health statuses depends on the used service discovery and health checking tools. Roughly speaking there are two different mechanics for this. Either the used service discovery system itself polls the service instance for health status at regular intervals, or the

service instance itself (or its dedicated sidecar proxy) pushes the health status to the service discovery system. (Arjis 2018; Checks n.d.; Richardson 2018b.)

3.4.1 External health checks

Keeping in mind that health checks can be considered to fall under black-box monitoring, it is reasonable to assume that it can sometimes be enough to inspect the service's health from outside of the service itself. Certain monitoring solutions such as Nagios with *http_check* plugin as well as service discovery solutions like Consul provide tools for querying service endpoints directly without having to modify the service itself. A simple agent querying a service's endpoints every now and then cannot provide much more information than whether the service endpoints are up or not; however, even that can go a long way. After all, a service instance not responding to endpoint requests is usually a clear sign that the service is not working as expected. (Checks n.d.)

While external checks are certainly better than nothing, there are some drawbacks. Since an external agent usually does not have access to the service's internals, the health checks have limited accuracy and may not be able to precisely pinpoint the root cause of failure. Thus, it can sometimes be better to have the service check its own health instead and expose the service's health status as a separate health check endpoint. Nevertheless, externally done health checks can still be valuable, and can be easier to add to pre-existing services than a dedicated health check endpoint. (Ewaschuk 2016.)

3.4.2 Health Check Endpoint

In the context of microservices, health checking usually refers to a health check endpoint that the service itself provides. This can also be referred to as a health check API. Instead of an external agent directly inspecting the service's endpoints, the service itself runs the required tests and checks to determine its own health. This can make the data provided by the check more accurate and makes inspecting dependencies easier, as the service itself should be aware of its own dependencies. (Arjis 2018.)

When designing a health check endpoint, it is important to keep in mind what kind of checks are required. Roughly speaking they can be categorized to three distinct groups: ones that only use resources local to the service, requiring no calls for other services; ones that make use of external resources such as downstream services as well; and ones that perform complete end-to-end transactions. One of the simplest health checks is to have an endpoint that responds with a static “OK” to every request it receives. While this only shows whether or not the service is up and running, and as discussed in chapter 3.3.1, there are cases where that is enough. If such a check is not responding, something is very clearly wrong. Additionally, such a check is relatively light weight as it only uses resources local to the service and serves static data. To expand on that, the health check endpoint could also check if the service’s other endpoints respond to requests in a valid manner, giving a more holistic look into the service’s health. (Arjis 2018; Richardson 2018b.)

Since many microservices can depend on external resources to function properly, it becomes important to know if the dependencies, such as databases or other services, are up and running. If a service registry is a part of the application’s infrastructure, checking it for healthy instances of the dependency service should be enough to make sure that there are healthy instances of the dependency available. However, ensuring that the dependency is functioning in the manner that the service expects may require the service to perform additional checks against an available instance of the dependency. This should catch most major unexpected API changes. (Arjis 2018.)

3.4.3 End-to-end checks

In addition to checking the health of single service instances, it can also be useful to create end-to-end health checks. An end-to-end health check executes a real business workflow to find any issues that simpler checks may have missed, and thus is often comprised of multiple different requests to multiple services. Since an end-to-end health check deals with real workflows, it is important to be careful with the data used. Depending on the used systems using real data can cause unfortunate side effects. By using fake data this check can be made *synthetic*, though the used data should still be selected with care. Another point to consider while designing

end-to-end health checks is that real business workflows usually take much longer to complete than simple endpoint pings. Because of this the frequency of these checks should be adjusted accordingly, especially if the workflows in question are resource intensive. (Arjis 2018; Newman 2015, 162.)

3.4.4 Health Check Severity

In addition to categorizing health checks based on their type, health checks can also be categorized by their severity. Generally, health checks can be put under two different severity labels of *critical* and *non-critical*. If a critical check does not pass, the service instance is marked to be in a critical state and should not be used by clients of the service. This state can then be displayed in the application's service registry, indicating that the service discovery system should not direct requests to the unhealthy service instances. (Arjis 2018; Checks n.d.)

It is useful to mark the checks that test the most crucial functions or dependencies of the service as critical, since if they are not working as expected, the service may be completely unable to serve its clients. Subsequently, if the subject that the check is testing is not crucial for the service to operate, it is advisable to mark the check as non-critical. If that check fails, the service instance can be flagged with a warning, signifying that while the instance may still be able to serve clients, its capability to do so has been reduced. For example, if a health check is used to monitor response latencies, when the latency reaches a certain threshold the check could be put into a "warning" state, marking the check as non-critically failing. If the service discovery system can take service health state into account, the service discovery system can know to reduce traffic directed to the unhealthy service instance, while still making use of it. As with critical checks, this can also be automated depending on the used service discovery system. (Arjis 2018; Checks n.d.)

4 Overview of used tools

4.1 Scala and Akka

Scala is a type safe multiparadigm programming language that implements features from both Object-Oriented Programming (OOP) and functional programming. Every value is an object, and every function is a value. Scala is conventionally run on top of Java Virtual Machine (JVM), but can also be compiled into JavaScript with Scala.js, or to native code with LLVM and Scala Native compiler (Scala Native n.d.). When run on JVM, Scala code is compiled into Java bytecode. The resulting bytecode can be run in any environment with Java Runtime Environment (JRE), where the bytecode is converted to native machine code by the JVM. The use of JVM allows seamless integration between Scala and plain Java. (The Scala Programming Language n.d.; Tour of Scala n.d.; Java Virtual Machine (JVM) & its Architecture n.d.)

Akka is a set of open-source libraries for Scala and Java. It is used to design systems that need scalability, resiliency, and distributability. It utilizes actor model to make the creation of concurrent, parallel, and distributed systems easier. This model is used universally across all Akka libraries. The model's consistent utilization within the Akka ecosystem enables in-depth integration between the different parts of the framework. (Introduction to Akka n.d.)

In a nutshell, Akka actors are units of execution in Akka. They execute tasks whenever a message is sent to them, provided that the actor has a task dedicated to the received kind of message. Akka actors are built on the concepts of actor model and event-driven model, making them well suited for asynchronous, event-driven applications. Akka actors and schedulers were chosen for the task as they would not require copious amounts of additional dependencies since the services were already built using Akka. (Akka Quickstart with Scala n.d.)

Scala and Akka were selected as the implementation tools for the proof of concept mainly to display that implementing features required by the thesis work was possible with technologies already used at Landis+Gyr. Other Scala frameworks, such as Spring or Play were considered as well during the preliminary planning. However, having another framework in addition to Akka would have been cumbersome, and

had the potential of conflicts between the frameworks. The proof of concept implementation produced in the study uses Scala version 2.12.6 with OpenJDK 8 which also contains the necessary JRE for running Scala code. The specific versions of all used Scala libraries can be found from the project's *build.sbt* files (see Appendix 1).

4.2 Consul

Consul is an MPL 2.0 licensed service mesh solution developed by HashiCorp (Dadgar 2013). Out of all the service discovery tools found during the preliminary phase of the thesis, Consul seemed to be the one providing the largest part of the features required by this study with easy implementation. It provides both HTTP and DNS interfaces that can be used for service discovery, and it supports health checks out of the box. However, various other tools such as Redis or Etcd were considered as well and would likely have been viable alternatives in other scenarios. The final proof of concept implementation produced during the study uses Consul version 1.4.3. (Introduction to Consul n.d.)

4.2.1 Consul cluster

To enable Consul to do service discovery as well as synchronization of service health information and other such data, Consul must be deployed in a cluster. Each member of that cluster must run the Consul Agent software to enable communications between them. The agent itself can be run in two different modes, those modes being *server* and *client*. Both serve crucial roles for the operation of the service discovery system. Cluster membership is handled using a gossip protocol provided by HashiCorp's Serf library that enables decentralized cluster membership and failure detection. (Consul Architecture n.d.)

HashiCorp recommends having either three or five server nodes per data centre. Different configurations can be used; however, with just one server data loss is bound to happen in failure cases, and having over five nodes is not ideal. This makes sure that the cluster's availability remains high even if some of the server nodes fail. This is because of the underlying *Raft* protocol that Consul uses for leader election. Raft protocol is used to achieve consensus about which server node should be

regarded as the leader node. For a server to be elected as a leader, it must receive a vote from more than half, a quorum, of the other server nodes. The server cluster is fault tolerant if this quorum can be reached. Therefore, with three server nodes, if one of the nodes fails, the cluster remains available if two of the remaining nodes are still working. In addition to the consensus quorum handling, the server nodes also take care of handling the state of the Consul cluster, as well as responding to client queries. (Consensus Protocol n.d.)

4.2.2 Service discovery

Consul's client agents are also part of the cluster. However, they do not participate in Raft. Instead, they forward requests from service instances to Consul server nodes (Consensus Protocol n.d.). The service instances using Consul mostly interact with the local Consul client agent without ever contacting server nodes directly. They are also responsible for running any health checks associated with their locally registered services. Services are registered to Consul using either its HTTP API or Consul configuration files. These *service definitions* should contain information required to connect to the service instance. At minimum, they contain the service's name, but other information, such as the port that the service uses is often good to have. Services can also be marked with tags that can be used as filers in certain queries. In addition, service definition can contain health checks for the service instance. Following the terms established in chapter 3.3, health checks defined with Consul are considered to be external health checks, though they can be used to poll the service's own health check endpoint should one exist. (Services n.d.; Checks n.d.)

4.2.3 Health checks

In addition to service discovery and mesh utilities, Consul also provides tools for executing a wide variety of external health checks at regular intervals. The simplest of them could be described as polls that just check if a network address is responding to requests. Especially polling checks done to HTTP addresses can synergize with service's own health check endpoint, since certain HTTP codes result in a different Consul service status. For example, HTTP response code 429 marks the service to be in "warning" state. (Checks n.d.)

The most interesting type of health check that Consul allows for is known as a “Time to Live” (TTL) health check. This means that the service itself pushes its own health status to Consul at regular intervals. If the service instance determines itself to be in a state where it is unable to serve clients, it simply relays the information to Consul, and the service instance is marked as unhealthy in Consul’s registry. Alternatively, if the service takes longer than expected to update its health, the instance is marked as critical. However, the one big issue with using Consul’s TTL health checks is that they heavily couple the service to Consul. While other service discovery and mesh solutions might support checks of similar fashion, coupling the service’s own health check logic to a service discovery system one can be a deal breaker if the service must be compatible with multiple different discovery systems. Additionally, if the timeouts and intervals of the TTL health check need adjusting, for full effect the value must be changed in the check’s definition as well as in the service itself. (Checks n.d.)

4.2.4 Other features

In addition to the features directly relevant to the study, Consul also contains many features that fall outside of the scope of this study but can nonetheless be useful. One of these features is Consul’s integrated key-value storage. It is separate from its service registry and can be used for configuration management, as well as other tasks where such a distributed storage could be of use. Another feature which looks to be of potentially high value is a feature called Connect. Consul Connect is Consul’s service mesh layer. Unlike how Consul is used for service discovery in this study, Connect is closer to server-side service discovery pattern. However, it also requires the use of the sidecar proxy pattern. When deployed using sidecar proxies the service itself can be unaware of the service discovery system. A Connect-aware sidecar proxy must be deployed for each of the service instances not natively integrating to Connect. Alternatively, the service could directly integrate with Consul, though that would highly couple the service with Consul. Connect provides automatic mutual TLS (mTLS) connection between the services that utilize it, as well as automatic certificate distribution, service identity and access control management, and *intentions*. Consul’s intention system is used to control which services can communicate with each other, which allows service segmentation

without complicated IP-based firewall rules. (Service segmentation made easy n.d.; Connect n.d.; Consul KV n.d.)

4.3 Verizon Helm

Verizon's Helm is a Scala library for interfacing with Consul's HTTP API. It serves a critical function in the implementation of this thesis work, as it is the primary way used to request service information from Consul's registry. Built on top of Cats functional programming library, it uses *Http4s* library to send requests to Consul's HTTP API. With Helm, operations such as registering services and health checks, and requesting service data is fairly simple. Each of the operations has a separate, pre-defined object that can be used to create a prepared request for the HTTP API. In complex applications, the resulting statement could be combined with further algebra thanks to Cats' *IO* data type. In the scope of this study, the operation is then run using Cats' *unsafeRunSync* function. (Baker 2018; IO n.d.)

4.4 Everpeace's healthchecks library

Healthchecks is a small health check library for Scala's Akka framework. It was created under an MIT licence by a man named Shingo Omura, also known by his GitHub handle Everpeace. It is an improved version of *Akka-HTTP Healthchecks* library created by a GitHub group known as *timeoutdigital*, containing additional features and improvements such as Kubernetes liveness/readiness probes. However, functions relating to Kubernetes were scoped out of this study. (Omura & Kato 2017.)

Everpeace's Healthchecks library allows its user to create health checks based on arbitrary Scala code, requiring only the code's end value is of correct type. Some additional restrictions are imposed by the two functions used to create check definitions. One is designed to be used with checks utilizing synchronous code, and the other utilizes Scala's *Future*-class to provide support for checks using asynchronous code. This nevertheless provides more than enough flexibility when it comes to the types of the checks. Thanks to this, creation of custom checks that make sure that dependencies and the service instance itself are working as expected

can be done with ease, especially if additional helper functions are created as templates for common checks. (Omura & Kato 2017.)

The defined checks can be included in the library's health check endpoint as either "fatal" or "non-fatal", signifying whether the service instance should be marked as critical if the check fails. Once the checks have been included in the endpoint, the service's health check status can then be accessed from the */health* HTTP endpoint. By default, the health status is only checked when the */health* endpoint is accessed. Whether this is a potential cause for issues falls outside of the scope of this study, however, certain additional measures could be implemented if the service's health needs to be monitored over time. (Omura & Kato 2017.)

4.5 Vagrant

Vagrant is a command line tool for building virtual machine environments and managing their lifecycles. Released by HashiCorp under MIT license (Hashimoto, Coder, Piedrafita, Rehm, Cain, & Mohammed 2019), it allows its user to quickly deploy pre-built virtual machines known as *boxes*. All Vagrant boxes must be run with a specific *provider* and are specific to the provider (Providers n.d.; Basic Provider Usage n.d.). Virtualization platforms such as VirtualBox and Hyper-V work as Vagrant providers to allow running and managing boxes. The default boxes provided by HashiCorp and Vagrant's community provide a good base for a development environment, especially since Vagrant allows the deployment and managing of multiple virtual machines at the same time. Parameters such as machines' network settings and start-up order can be further tweaked with a *Vagrantfile* before deploying the machines (Vagrantfile n.d.). However, since even after Vagrantfile tweaks the machines are rarely in a perfect shape, Vagrant's *provisioning* feature allows for additional configuration management. This way a separate configuration management system such as Ansible or Chef can be used to further set up the machines during the usage of the Vagrantfile. Thanks to this, software installations and other needed tasks can be done within the target virtual machine automatically once the configuration management system is set up (Provisioning n.d.).

In the scope of this study, two of the provision options prevail. Since configuration management systems were not in the study's scope, using simple shell script and file-based provisioning options was found to be a better use for resources. However, in production proper configuration management systems will likely prove to be more effective and maintainable. Vagrant's "File" provisioner allows the simple copying of files and directories from the host machine to the guest machines. This way each microservice's code bases are copied to appropriate guests. The "Shell" provisioner allows the execution of series of shell commands on the guest machine, defined either as an inline string or in an external file. (Provisioning n.d.)

The final proof of concept environment was tested with Vagrant version 2.2.4. The virtual machines in the environment are based on the Debian/stretch64 Vagrant box for VirtualBox, with the used box version being 9.8.0.

4.6 Oracle VirtualBox

VirtualBox is a GPLv2 licensed virtualization platform developed and released by Oracle. It is designed to run virtual machines in both enterprise as well as home use. Additionally, it can be used as a Vagrant provider, allowing Vagrant to deploy and run virtual machines on top of VirtualBox. These virtual machines (VM) can be referred to as *guests operating system*, and the machine they are running on is referred to as a *host operating system*. VirtualBox was selected as the platform for running the virtual machine cluster necessary for running the proof of concept of this study mostly due to it being already familiar and easily available thanks to its open source license. Version 6.0.4 was selected for the project as it was the latest version available. (Oracle VM VirtualBox n.d.; Provisioning n.d.)

4.7 Sbt

Sbt is an interactive command line tool for building and running Scala applications. Published by Lightbend Inc. under BSD a license (Licenses n.d.), it makes compiling Scala projects a trivial task. It was selected for this thesis because code examples used as a base for the work were created using it. Sbt also handles dependency management via its build files, and it also has support for several testing frameworks

and custom build tasks (Task graph n.d.). While Sbt supports building and running Scala applications out of the box, a plugin called Sbt-revolver was taken into use to help streamline the development process. With it, a running Scala application can be restarted without having to interrupt Sbt's own interactive shell (Rudolph, Mathias, Bo, Rogach, Engelen, ReadMeCritic, Regadas, Aleksandrov 2017). Other build tools such as Gradle or Maven could have been used as well, however, as pre-made service templates used in the project were built with Sbt instead, it was selected as the build tool for the proof of concept. Converting the templates away from Sbt would have had very little return on investment within the duration of this project, as in the project's scale Sbt seemed to have no significant drawbacks. However, in production investigating tools other than Sbt might be useful, as Sbt has been criticized for poor performance by some parties such as Vinyas Maddi (2018). The final proof of concept produced in the study uses Sbt version 1.2.7. (sbt Reference Manual n.d.)

5 Development plan

5.1 Preface

The development plan for the work started out as a "plan-as-you-go" type of deal with only a vague end goal in mind: produce a few microservices that utilize some sort of service discovery system and implement health check APIs. As the subject matter was very unfamiliar, it was deemed more important to dive into studying the core concepts before setting too much of the technical stuff into stone. Understanding the core concepts was exceptionally crucial, since the ideas and principles behind service discovery were held in a higher value than the specific technologies. However, some technical requirements were clear from the start: the solution would have to use Scala as its primary programming language, and the code would need to be compatible with Scala's Akka framework. This imposed some restrictions on what other libraries or frameworks could be viably used for the implementation. Although alternative frameworks certainly looked intriguing on some fronts, in order to prevent potential conflicts with Akka, the use of other large libraries and frameworks would be kept as low as attainable.

5.2 Proof of Concept services

5.2.1 Basic service template

To demonstrate service discovery properly, more than one microservice would be needed. These services would be created using Scala with Akka framework and would be based on simple Akka HTTP application templates. These templates were found in GitHub. Gabriel Francisco, also known by his GitHub handle *gabfssilva*, has created and licensed multiple Akka HTTP microservice templates under the open source MIT license (2017). As some sort of database support was included as an optional stretch goal in the scope of the project, the template with premade MongoDB database support would be selected as a base. Since the service template was built using Sbt, it was selected as the build tool for the thesis's microservices. The template also comes with a simple health check endpoint already included. However, a separate health check library is used instead since the template's health check endpoint is too simple to provide adequate amount of value for the proof of concept.

If attainable, services would be containerized for better distributability as a nice-to-have feature. This could be left out of scope if need be. However, even if a container solution was used, container-exclusive technologies such as Registrator (n.d.), which would allow automatic service registration to Consul on service container start up, will not be used. This is to ensure that the designed system can be used as a proper reference even if the target application does not use containers.

5.2.2 Service architecture

To keep the system's structure simple enough, an architecture with one "client" service and one "server" service was selected. This architecture would also fulfil requirement NR-01 defined in chapter 2.2. The server service would be deployed in multiple instances. It would output pseudorandom numbers at an HTTP endpoint with no arguments or parameters required. This service would be named RNG-Service. The client service, on the other hand, has an HTTP endpoint which is supplied with a number. The service then finds an instance of RNG-Service with the help of the used service discovery system. After a healthy instance is found, a

random number is fetched from it. The received number is then multiplied with the number supplied to the service and returned to the user. Its name would be Multiplication Service. An additional stretch goal feature for Multiplication Service would be to have each multiplication event and their results stored into a database, and an HTTP endpoint which could be used to display the results from previous multiplication events. The database would also be registered as a service to the discovery system. This feature was left out of the final scope of the proof of concept as the project's goal became clearer along with deeper understanding its subject matters, which caused the scope of the entire study to become more tightly controlled.

5.2.3 Service health checks

Both services would implement a health check HTTP endpoint to fulfil the requirement FR-02 as defined by chapter 2.2. For the purposes of this proof of concept quite simple health checks should suffice. At minimum, services should check if their own endpoints are available. Checking that their dependencies are also running and healthy provides additional insight into whether the service can function properly. Thus, Multiplication Service should check for RNG-Service's state during its own health checks. Checking the status of external services can be done by requesting for a healthy service instance from Consul by using Helm. These checks would be moderately easy to implement with plain Scala and Akka HTTP, but if a suitable Scala library is found for that purpose, it will be used provided that it is compatible with other used technologies and its license allows for it. If no such library is found, the health check endpoint will be purpose built for each of the proof of concept microservices.

Since Everpeace's *healthchecks* Scala library was found during the preliminary research and was found to be adequate for the purposes of this thesis, it would supersede any proprietary, self-made attempts at a health check API. The HTTP endpoints used for health checking would run the checks assigned to them whenever the endpoint was requested. This way the information is always fresh, but there's a risk that the health at the second of the request is not properly indicative of the service's overall health. For example, if there's an issue where the service

unexpectedly restarts itself automatically at regular intervals, and the service's health is checked with the same interval, the service will still seem healthy if the restart and check timings do not align properly.

The route for the endpoint should optimally be just a sub-directory under the service itself, such as */health* or */api/health*. The data returned by the checks would ideally be in a JSON format, with each check having its own JSON object compiled under a larger summary object. Depending on the severity and the status the checks, the data would be returned with a different HTTP status code. Passing checks should produce code 200 OK, whereas fatally failing checks would be returned with a code 503 Service Unavailable.

5.3 Utilizing service discovery

The service discovery used in the proof of concept would be HashiCorp's Consul. The services would query Consul's HTTP API for healthy instances of their dependencies utilizing the client-side service discovery pattern. In order to demonstrate different ways to register services with Consul, RNG-Service instances would use the HTTP API to register themselves on service start-up, while Multiplication Service would be registered using a Consul service definition. For a proper demonstration, six virtual machines would be deployed. Three of them would serve as Consul server nodes, and three others as Consul client nodes. One of the client nodes would run an instance of Multiplication service, while the other two would be running an RNG-Service instance each. The virtual machines would be deployed and configured using Vagrant and VirtualBox. Vagrant could also be used to package the service environment machines into pre-built packages, so that deploying the proof of concept set-up in a clean environment could be done more easily.

Consul would also be used to run external health checks on the service instances. RNG-Service's health endpoint would be regularly queried for fresh health check data by the service's local Consul agent, and the service's status would be updated to Consul's registry accordingly. Multiplication Service on the other hand would update its status to Consul actively. At regular intervals the service queries its own health endpoint for health data and pushes the appropriate health status to its local Consul

agent. If Multiplication Service fails to update its state within the appropriate time window, the service is marked as failing.

5.4 Stretch goals

An additional stretch goal feature would be the creation of a .NET implementation of the system. As with other optional “nice-to-have” features included in the project, it is to be left out of the scope if the scope is deemed to be too wide and the features could not be feasibly implemented in a sensible time frame. However, even if no .NET implementation is created, the systems functionality should not be too reliant on Scala’s special features to ensure that the results could be replicated or mimicked with other programming languages as well. This may impose further restrictions to the used technologies. For example, the service discovery system should be usable with microservices made with different languages. Because of this and because of additional time cost associated with getting familiar with new technologies, Akka Management will not be used for service integration with Consul. However, if the service discovery system is accessible via an HTTP API or something similar, it should be usable with a good majority of modern languages and frameworks. Alternatively, a language-agnostic sidecar service instance could be used as a proxy between the service instance and the service discovery system.

Even if a database recording multiplication events is not implemented, registering some external service that is in the control of some party could also provide value on a demonstrative level. Sometimes a service can depend on another service that is outside of the control of the developer, e.g. a map or a geographical API. Another optional feature that would be nice to have in the proof of concept would be to implement a Time to Live (TTL) based Consul health check. Since Consul has support for those, adding a timed check providing relevant health check data to Consul using Akka’s actors and schedulers should be attainable. In the constraints of the study a very rudimentary TTL check should still demonstrate their functionality quite well.

6 Implementing the proof of concept system

This chapter explains how the proof of concept system for the study was implemented. The source codes of the proof of concept can be found from Git repositories linked in Appendix 1.

6.1 Creating a Consul server cluster

The creation of the proof of concept system started out by setting up the virtual machines required for it. These virtual machines were preconfigured and launched using a Vagrantfile modified from HashiCorp's Consul guide (see Freddy 2018). The modifications introduce four additional virtual machines on top of the two machines already defined in the file as well as major additions to provisioning, such as OpenJDK and Sbt downloads. The exact Vagrantfile can be found through Appendix 1 along with the rest of the proof of concept's code. As planned, three virtual machines were created to serve as Consul server nodes, and three others were created to serve as Consul client nodes. The server nodes would serve as the backbone of the Consul cluster, whereas the client nodes would later be used to run the proof of concept services as along with Consul's client agent.

Each of the server nodes is configured in a similar fashion with only difference being the used IP addresses. While the syntax used for configuration is JSON, Consul also supports the use of HashiCorp's own HCL (HashiCorp Configuration Language) syntax for its configurations. Figure 1 shows the complete configuration of the second server node. In order to get Consul to recognize the virtual machines as server nodes, the server parameter is configured to be *true*. On the client nodes this field does not need to be specified. Setting the *bootstrap_expect* parameter as three guarantees that Consul waits until that amount of server nodes is available before attempting leader election. This value should be the same on all server nodes or not existent at all to prevent inconsistencies and cluster states where multiple servers consider themselves to be the leader node. The initial joining of these server nodes is known as *bootstrapping*. (Bootstrapping a Datacenter n.d.; Configuration n.d.)

```
{
  "server": true,
  "data_dir": "/tmp/consul",
  "bootstrap_expect": 3,
  "node_name": "server-02",
  "bind_addr": "172.20.20.21",
  "retry_join": ["172.20.20.20"]
}
```

Figure 1. Consul server node configuration

Another important Consul server configuration parameter is the IP address in the *retry_join* field which can also be seen in Figure 1. This parameter can be supplied with an array of IP addresses. When the Consul agent is started, it attempts to join other Consul members using the addresses listed in the array. The joining is retried until it is successful, which can be useful when starting up multiple Consul nodes at roughly the same time. Joining can also be done without retries; however, without retries the join is only ever attempted once when the Consul agent is started. For the sake of easier configuration and deployment, every node in this study has been configured to join the first server node (172.20.20.20). The joining behaviour can be seen in Figure 2, where joining is seen from the third Consul server node's perspective. In this case, the first two server nodes are already running, and the third node is started after them. The figure also displays cluster bootstrapping and cluster leader election as indicated by the arrows in the figure. (Adding & Removing Servers n.d.; Configuration n.d.)

```
vagrant@s3:~$ consul agent -config-dir=/etc/consul.d
bootstrap_expect > 0: expecting 3 servers
==> Starting Consul agent...
==> Consul agent running!
           Version: 'v1.4.4'
           Node ID: '[redacted]'
         Node name: 'server-03'
        Datacenter: 'dc1' (Segment: '<all>')
            Server: true (Bootstrap: false)
       Client Addr: [127.0.0.1] (HTTP: 8500, HTTPS: -1, gRPC: -1, DNS: 8600)
      Cluster Addr: 172.20.20.22 (LAN: 8301, WAN: 8302)
           Encrypt: Gossip: false, TLS-Outgoing: false, TLS-Incoming: false

==> Log data will now stream in as it occurs:

    2019/04/14 12:32:24 [INFO] raft: Initial configuration (index=0): []
    2019/04/14 12:32:24 [INFO] raft: Node at 172.20.20.22:8300 [Follower] entering Follower state (Leader: "")
    2019/04/14 12:32:24 [INFO] serf: EventMemberJoin: server-03.dc1 172.20.20.22
    2019/04/14 12:32:24 [INFO] serf: EventMemberJoin: server-03 172.20.20.22
    2019/04/14 12:32:24 [INFO] consul: Adding LAN server server-03 (Addr: tcp/172.20.20.22:8300) (DC: dc1)
    2019/04/14 12:32:24 [INFO] consul: Handled member-join event for server "server-03.dc1" in area "wan"
    2019/04/14 12:32:24 [INFO] agent: Started DNS server 127.0.0.1:8600 (udp)
    2019/04/14 12:32:24 [INFO] agent: Started DNS server 127.0.0.1:8600 (tcp)
    2019/04/14 12:32:24 [INFO] agent: Started HTTP server on 127.0.0.1:8500 (tcp)
    2019/04/14 12:32:24 [INFO] agent: started state syncer
    2019/04/14 12:32:24 [INFO] agent: Retry join LAN is supported for: aliyun aws azure digitalocean gce k8s os p
acket scaleway softlayer triton vsphere
    2019/04/14 12:32:24 [INFO] agent: Joining LAN cluster...
    2019/04/14 12:32:24 [INFO] agent: (LAN) joining: [172.20.20.20]
    2019/04/14 12:32:24 [INFO] serf: EventMemberJoin: server-02 172.20.20.21
    2019/04/14 12:32:24 [INFO] serf: EventMemberJoin: server-01 172.20.20.20
    2019/04/14 12:32:24 [INFO] consul: Adding LAN server server-02 (Addr: tcp/172.20.20.21:8300) (DC: dc1)
    2019/04/14 12:32:24 [INFO] agent: (LAN) joined: 1 Err: <nil>
    2019/04/14 12:32:24 [INFO] agent: Join LAN completed. Synced with 1 initial agents
    2019/04/14 12:32:24 [INFO] consul: Found expected number of peers, attempting bootstrap: 172.20.20.22:8300,17
2.20.20.21:8300,172.20.20.20:8300
    2019/04/14 12:32:24 [INFO] serf: EventMemberJoin: server-02.dc1 172.20.20.21
    2019/04/14 12:32:24 [INFO] serf: EventMemberJoin: server-01.dc1 172.20.20.20
    2019/04/14 12:32:24 [INFO] consul: Handled member-join event for server "server-02.dc1" in area "wan"
    2019/04/14 12:32:24 [INFO] consul: Handled member-join event for server "server-01.dc1" in area "wan"
    2019/04/14 12:32:24 [INFO] consul: Adding LAN server server-01 (Addr: tcp/172.20.20.20:8300) (DC: dc1)
    2019/04/14 12:32:29 [INFO] agent: Synced node info
    2019/04/14 12:32:29 [INFO] consul: New leader elected: server-01
```

Figure 2. Consul join behavior (Node ID redacted)

The rest of the parameters shown in Figure 1 are not as important in the perspective of this study. The *data_dir* parameter points to a local directory and is required for storing the state of the Consul agent, which is especially important on server nodes for preserving cluster state. The targeted directory should be persistent across reboots. This directory can also be used to store various access tokens if Consul's ACL feature is taken into use, in which case restricting file access permissions in that directory should be considered.

In addition to the IP in the *retry_join* field, another address can be seen in the *bind_address* field. On agent start-up, Consul binds to this private, locally available IPv4 address. If the node only has one available private IPv4, this field can be left empty, as Consul defaults to the only address available. However, if multiple addresses can be used, the field must exist with a clearly defined value, otherwise Consul exits with an error. Comparatively the *node_name* field has a much simpler purpose. As one might expect from the field's name, it is used for naming the Consul nodes. The node's name can be used in various Consul registry queries e.g. listing all

the available services for the node, and the name must be unique within the Consul cluster. (Catalog HTTP API n.d.; Configuration n.d.)

With a configuration like this, starting up a server node is a trivial task. The only additional required parameter that must be supplied to Consul on startup is the location of the directory containing Consul's configurations. In this case the configuration files are put into the */etc/consul.d* directory on each of the virtual machines. The Consul agent is started in the same way on every virtual machine of this study: first each of the machines is connected to with a separate SSH instance using *vagrant ssh* using the machine's hostname as the connection parameter, after which the command *consul agent -config-dir=/etc/consul.d* starts up the agent. The configuration files take care of the rest. (Configuration n.d.)

If there are multiple configuration files in the target directory, the files are loaded in an alphabetical order. Alternatively, if the configuration files are not located in a dedicated configuration directory, *-config-file* command line argument could be used to specify a certain configuration file instead. The agent's configuration parameters assigned in the configuration files could also be supplied as command line flags just like *config-dir*, or as environmental variables. However, using configuration files instead was found to be more convenient in the long run. If a parameter is assigned in more than one of those places, the last one loaded takes precedent by default. Thus, configuration files override environmental variables, and environmental variables override command line parameters. Deploying the Consul server nodes along with the rest of the Consul nodes is explained in more detail in chapter 6.4. (Configuration n.d.)

6.2 RNG Service

After the Consul server nodes were up and running, it was time to start creating the microservices that would utilize the service discovery system. As discussed in the development plan, both services are be based on a Scala Akka HTTP service template (see chapter 5.2.1). Since the RNG Service would have no dependencies on any other services, the development efforts were started from that.

6.2.1 Handling dependencies

The purpose of the RNG Service was to generate pseudorandom numbers at an HTTP endpoint. In order to create the service, the source files of the service template were copied to a new local folder. This folder would serve as the home for the RNG Service. The very first file in need of modification is the *build.sbt* file in this folder. This file must be modified any time a new library is included into the project, or if some other settings related to Sbt need to be changed (Build definition n.d.; Library dependencies n.d.). Additionally, any Sbt plugins such as Revolver must be specified in a separate *plugins.sbt* file under the *project* folder contained in the project's root folder.

In this case the project's Sbt files contained in the template already had most of the dependencies required by the proof of concept services such as Sbt-revolver plugin and Akka HTTP library along with a few other Akka libraries. However, in order to take the plugins needed for Consul integration and health checks working, they must be added to *build.sbt* as library dependencies. The required libraries are *healthchecks-core* from the *everpeace* package, and *http4s* from Verizon's *helm* package. These can be seen in Figure 3. However, in order to get Sbt to find the *everpeace* package, an additional maven repository must be added to Sbt's resolvers. Additionally, the Fakemongo package is added as a normal dependency in addition to the pre-existing test dependency as the lack of a local MongoDB database started causing issues while trying to run the template project. However, all the template's MongoDB usage was later removed as redundant.

```
// ...
resolvers += Resolver.bintrayRepo("everpeace","maven")
// ...
libraryDependencies ++= Seq(
  // ...
  // probably not necessary in the end
  "com.github.fakemongo" % "fongo" % "2.1.0",

  // Everpeace Healthcheck https://github.com/everpeace/healthchecks
  "com.github.everpeace" %% "healthchecks-core" % "0.4.0",

  // Verizon Helm https://github.com/Verizon/helm
  "io.verizon.helm" %% "http4s" % "4.0.0",
  // ...
)
// ...
```

Figure 3. RNG Service's *build.sbt* changes from template base

6.2.2 Implementing Random Number Generator endpoint

After the necessary plugins and libraries were added to the service's dependencies, it was time to implement the service's core functionality. The RNG endpoint would replace the template's User endpoint. As the logic of the pre-existing endpoint was far more complicated than was required for a simple random number generator, most of the endpoint's original code was removed. What was left is a HTTP route *api/rng* which returns a random number between zero and nine. As can be seen in Figure 4, this is done using Akka HTTP for the request handling and Scala's *Random* utility for generating the random number itself.

```
// ...
class RNGEndpoint {
  val rng = new Random
  val rngRoute: Route = {
    pathPrefix("api" / "rng") {
      get { complete(rng.nextInt(10).toString) }
    }
  }
}
```

Figure 4. RNG Endpoint for generating random numbers

The final handling of all HTTP routes of the service is done in the main application itself. The service's pre-defined endpoints are first gathered under a single *Endpoints* class acting as a container of sorts. These endpoints include the *RNGEndpoint* responsible for the primary function of the service, as well as the redundant bare-bones health check endpoint included in the service template. The *Endpoints* class also handles logging of the statuses and paths of any sent HTTP responses. This *Endpoints* container class is then initialized along with the routes within it in a module class *AllModules* by using the template's included *Macwire* library to inject the required Akka objects into the into the route classes (Warski 2019).

The service template already has most of the endpoint management configured, and thus the changes required in the *Endpoints* and *AllModules* classes are limited to the class name of the service's primary endpoint, in this case *RNGEndpoint*. The module class is then imported in the service's main application object called *Application*. Once the module has been imported, its routes are taken into use by "binding" the routes to a locally available IP address and port. The importing and binding of the routes is displayed in Figure 5. In this case all the service's own routes are bound to port 8080 using the address 0.0.0.0. With this address the service listens to every network interface available on the machine (Iwaya 2015). The binding also contains routes related to the used health checks library. The usage of the health check library is explained further in a later paragraph.

```
// ...
import modules.AllModules
// ...
object Application extends App {
  val modules = new AllModules
  // ...
  import modules._
  // ...
  Http().bindAndHandle(
    HealthCheckRoutes.health(rngCheck, rngCheck2, googleCheck) ~
    modules.endpoints.routes, "0.0.0.0", 8080).onComplete {
    case Success(b) => system.log.info(
      s"application is up and running at
${b.localAddress.getHostName}:${b.localAddress.getPort}")
    case Failure(e) => system.log.error(
      s"could not start application: {}", e.getMessage)
  }
}
```

Figure 5. Binding and handling the HTTP routes

6.2.3 Health check endpoint

After binding the service's main endpoints to a port, it was time to take the health checks library into use. This can be done with ease, as the health checks can run any kind of code that that has the end type of either of the library's specialized health check statuses, *healthy* or *unhealthy*. The health checks can be defined to be either synchronous or asynchronous. As most of the checks in the service were using asynchronous HTTP calls to verify health status, the asynchronous version of the health check definition saw the most usage. Figure 6 displays one of the simple HTTP checks that the service uses to determine its own health. It sends a simple get request to its own RNG endpoint and expects a successful HTTP response code. In addition to a name, the check can also be supplied with an optional severity indicator which defaults to "Fatal". A failing fatal check indicates that the service is not capable of serving clients at the moment.

```
  val rngCheck = asyncHealthCheck(name = "rng health check") {
    val rngEndpointFuture: Future[HttpResponse] = Http()
      .singleRequest(HttpRequest(uri = "http://127.0.0.1:8080/api/rng"))
    // Mapping Example 5,
https://www.programcreek.com/scala/akka.http.scaladsl.model.HttpRequest
    rngEndpointFuture.map {
      response =>
      if (response.status.isSuccess) healthy
      else unhealthy("RNG endpoint not available")
    }
  }
```

Figure 6. Simple HTTP health check using healthchecks library

To make creating simple HTTP request-based checks easier, a helper function was created to automatically construct an *asynchHealthCheck* objects with just a check name, target address, and an optional severity parameter. As can be seen in Figure 7, checks using this *httpHealthCheck* function are functionally nearly identical to the check displayed in Figure 6. Both use Scala Futures and Akka's HTTP client to make a simple HTTP GET request to the targeted address, and the evaluation of success and failure is the same as well.

```
  def httpHealthCheck(
    name: String,
    target: String,
    severity: Severity = Severity.Fatal
    ) = asyncHealthCheck(name = name, severity = severity){
    val endpointFuture: Future[HttpResponse] = Http()
      .singleRequest(HttpRequest(uri = target))
    // Mapping Example 5,
https://www.programcreek.com/scala/akka.http.scaladsl.model.HttpRequest
    endpointFuture.map {
      response =>
      if (response.status.isSuccess) healthy
      else unhealthy("${target} not available")
    }
  }
  // ...
  val rngCheck2 = httpHealthCheck(
    "rng check 2", "http://127.0.0.1:8080/api/rng"
  )
```

Figure 7. Helper function for creating health checks

For this case only three checks were made using the *healthchecks* library. Two pinging the service's own RNG endpoint and one pinging Google's front page to ensure that the service has internet connection. Two other checks that randomly switched between passing and failing states were designed and failed, but as the service grew larger those were dropped out as their unpredictability started causing frustration when developing and tweaking the other features of the proof of concept. After all the checks were defined, they as well were bound to the same port as the other endpoints of the service by bundling them together using the *HealthCheckRoutes.health()* function provided by the library. This is displayed in Figure 5 along with the binding of other service endpoints. After binding, the health check information is available from the service's */health* subdirectory. The health status is checked every time the endpoint is called. By default, the endpoint only provides an HTTP status code; 200 if the service is healthy and no fatal checks are failing, or 503 if there are any fatal fails. However, if the endpoint is supplied with a get parameter *full=true*, the exact state and output of each check can be seen in detail. (Omura & Kato 2017.)

The template's pre-existing health check endpoint was also modified to fit the service, even though it had no real use or purpose at this point. While the original health check endpoint included in the template only returns a static "OK" status with no actual checks, the modified endpoint is directly comparative to the check definition shown in Figure 6. After modifications, the endpoint sends a simple GET request to the service's RNG endpoint to verify that the endpoint is up and running. The endpoint it available under the service's */api/health-checks* path and brings hardly any additional value to the health checks made with the *healthchecks* library.

6.2.4 Consul integration

After the service had its health check endpoint up and running, it still needed to be integrated with Consul in order to be utilized in multiplication service. Before the service itself can be locally registered, the Consul node itself must be set up and configured. The RNG service is deployed on two Consul client nodes. Their configuration is very similar to the ones of server nodes (see Figure 1). However, the *server* parameter is assigned to be *false*, and *bootstrap_expect* parameter is

completely removed. The configuration file is placed in a folder with the same path as on the server nodes, and the Consul agent is started in the exact same way as well.

The service is registered to Consul using Verizon's *Helm* library. As the service was to have Consul health checks associated with it, those checks were defined first. Helm provides a case class called *HealthCheckParameter* for creating Consul health check deifinitions, allowing the assignment of most health check parameters supported by Consul. However, when it comes to health checks, certain discrepancies can be seen when comparing Helm and Consul's HTTP API. For example, Helm does not currently have support for gRPC based checks or header parameters for HTTP based checks. Fortunately, this does not impede the progress of the proof of concept, as most of the service checks work with HTTP calls.

However, issues arise from the fact that none of *HealthCheckParameter*'s parameters have a default value, meaning that each of the 13 parameters must be assigned an explicit value, even if that parameter is of no use for the check in question. While this might be a deliberate design choice by Verizon, it is still a major inconvenience when creating health checks. Thus, a helper function was made for creating *HealthCheckParameter* objects without the need to assign each parameter manually.

The *easyHealthParam* helper function automatically assigns *None* as default value for every parameter other than the check name, as it is required by Consul. The function definition was added into a separate *Helpers* object, which was then imported in the main *Application* object. This function was then used to define two HTTP checks for RNG Service; one to check on the RNG endpoint and the other to check the health check endpoint. Both checks would run every 15 seconds and were marked as "Critical" for Consul, meaning that if either of the checks was failing, the service instance is marked as unhealthy in Consul's registry. Example usage of *easyHealthParam* can be seen in Figure 8. As every parameter apart from the service's name is defined to use Scala's *Option* class, each of the values had to be wrapped inside Scala's *Some* case class.

```
import Helpers._
val endpointHealthCheck: HealthCheckParameter = easyHealthParam(
  name = "RNG endpoint check",
  http = Some("http://127.0.0.1:8080/api/rng"),
  interval = Some(Interval.Seconds(15)),
  notes = Some("Created via Helm"),
  initialStatus = Some(HealthStatus.Critical)
)
```

Figure 8. Using the *easyHealthParam* function to create a HTTP check for Consul

Once the Consul health checks for the service have been defined, they can be added to a Consul service definition along with the rest of the service's details. In this case the service definition consists of a service name, service port, and a list of Consul health checks associated with the service (see Figure 9). The registration command is then run on service start-up right before binding the service's endpoints by using *Cats* library's *unsafeRunSync* function, making the service available for discovery. However, as the registration command is not wrapped with any kind of error handling, the service crashes on start-up if local Consul agent is not available.

```
// service, id, tags, address, port, enableTagOverride, check, checks
val registerCommand: IO[Unit] = helm.run(
  interpreter, ConsulOp.agentRegisterService(
    "rng-service", None, None, None, Some(8080), None, None,
    Some(NonEmptyList.of(endpointHealthCheck, primaryHealthCheck))
  )
)
// If Consul is not available when this command is run, the whole
service fails to start.
registerCommand.unsafeRunSync()
```

Figure 9. Registering RNG service to Consul

6.2.5 Running the service

After the service's code is done, it can be finally deployed to the proof of concept environment. Using the same Vagrantfile as the three Consul server nodes, two other virtual machines are started up. As with Consul server nodes, these two machines automatically receive the needed Consul agent configuration as JSON files. The service's code is also copied onto the machines. Both machines were connected to

with two SSH connections per machine. Each one of the SSH connections is created with the command *vagrant ssh* on the host machine with the machine's hostname as the parameter. The first connection was used to start the Consul agent with the same command as with Consul server machines; *consul agent -config-dir=/etc/consul.d*. While a functioning Consul server cluster is not absolutely necessary for the client node to function, it can nevertheless be recommended if only to reduce the amount of errors in Consul agent's output. However, as mentioned in the previous chapter, the local Consul agent is required for the service to start.

After the client agent is up and running, the second SSH connection is used to navigate to the directory containing the service's code, in this case */home/vagrant/rng-proto*. Sbt shell is then started within that directory with the command *sbt*. Within the Sbt shell the service is compiled and run with sbt-revovler's command *reStart*. Alternatively, *run* command could be used in its stead, however, *reStart* allows the use of *reStop* for stopping the running service without exiting the Sbt shell. In the proof of concept set up it is crucial to leave the SSH connections open until the service is to be terminated, as closing the connection kills any processes assigned to the SSH session.

After starting the service, it is available in Consul registry. This can be demonstrated by making a simple query to Consul's HTTP API on any of the machines in the Consul cluster. This, along with the function of the service's RNG endpoint is displayed in Figure 10. In the figure, one of the server nodes requests the service from Consul's service catalog using Curl. After that, the service's RNG endpoint is requested multiple times to demonstrate that the endpoint can be reached from its advertised address. In a completely deployed proof of concept environment the catalog request would display two available endpoints for RNG Service. For the sake of brevity Figure 10 was produced with only one deployed RNG Service instance.

```
vagrant@s2:~$ curl 127.0.0.1:8500/v1/catalog/service/rng-service -s | jq
[
  {
    "ID": "[redacted]",
    "Node": "client-02",
    "Address": "172.20.20.11",
    "Datacenter": "dc1",
    "TaggedAddresses": {
      "lan": "172.20.20.11",
      "wan": "172.20.20.11"
    },
    "NodeMeta": {
      "consul-network-segment": ""
    },
    "ServiceKind": "",
    "ServiceID": "rng-service",
    "ServiceName": "rng-service",
    "ServiceTags": [],
    "ServiceAddress": "",
    "ServiceWeights": {
      "Passing": 1,
      "Warning": 1
    },
    "ServiceMeta": {},
    "ServicePort": 8080,
    "ServiceEnableTagOverride": false,
    "ServiceProxyDestination": "",
    "ServiceProxy": {},
    "ServiceConnect": {},
    "CreateIndex": 2356,
    "ModifyIndex": 2356
  }
]
vagrant@s2:~$ curl 172.20.20.11:8080/api/rng -s | jq
5
vagrant@s2:~$ curl 172.20.20.11:8080/api/rng -s | jq
8
vagrant@s2:~$ curl 172.20.20.11:8080/api/rng -s | jq
5
```

Figure 10. Requesting RNG service from Consul catalog (Node ID redacted)

6.3 Multiplication Service

The multiplication service was quite different in nature from RNG service. However, as a significant amount of the design is still similar between RNG and Multiplication services, RNG Service was used as a base for Multiplication service. This chapter will highlight the differences between them.

6.3.1 Handling dependencies

The dependency handling in Multiplication Service is done in the exact same way as in RNG service. However, as the Multiplication Service also implemented a Time to Live Consul health check using scheduled actors, Akka's Actors functionality was needed as an additional dependency. Additionally, since Verizon's Helm lacks support for registering external services, Consul's HTTP API needed to be dealt with directly in applicable cases. Thus, a JSON library was required. While the dependencies already contained *Circe* JSON library, *akka-http-spray-json* was adopted as its alternative for creating JSON objects needed for the registration of an external service. These added libraries can be seen in Figure 11. However, the Sbt plugins used in the service are identical to the ones used in the RNG Service.

```
// ...
libraryDependencies ++= Seq(
  // ...
  "com.typesafe.akka" %% "akka-http-spray-json" % "10.1.5",
  "com.typesafe.akka" %% "akka-actor" % akka,
  // ...
  "io.spray" %% "spray-json" % "1.3.5",
  // ...
)
// ...
```

Figure 11. Multiplication service's dependency additions

6.3.2 Implementing Multiplication endpoint

Although Multiplication Service's dependencies are very similar to those of RNG Service, their primary endpoints are vastly different. While RNG Service's primary endpoint simply responds to any GET request with a random number, Multiplication Service handles requests in several steps. Upon requesting Multiplication service's path along with an integer (e.g. */multiply/3*), the service first attempts to discover healthy RNG Service instances from Consul via Helm. After Consul has returned a list of healthy RNG service instances, the list is inspected to see how many instances are available. If no available RNG endpoints are returned, the request is responded with a HTTP status code 503, indicating that the Multiplication service is currently

unavailable, along with a response message indicating that the reason for unavailability is that the service's dependency is unavailable. Alternatively, if one or more RNG endpoints are available, one of them is selected at random. However, if the endpoint was not supplied with an integer, the request is immediately responded to with HTTP status code 400, indicating that the request is bad and cannot be fulfilled.

After Multiplication service has selected an RNG Service instance, that instance's RNG endpoint is sent a simple HTTP GET request using Akka HTTP's HTTP client. The response entity is converted to a string via Akka HTTP's Unmarshalling feature. The resulting string is then parsed to an integer inline and multiplied with the integer supplied to the Multiplication endpoint. The resulting value is then sent to the client using the Multiplication endpoint.

The random selection of the RNG Service instance served as a make-shift load balancer of sorts and made sure that both RNG Service instances receive traffic. If no random selection was done, one instance of the RNG Service would have received all the traffic as Consul returns the service instances in the same order on every request. In the proof of concept environment, the order of the returned service instances remains the same even if the instances were sorted by latency, as the stability of the environment's network never caused any noticeable latency fluctuations between the different Consul nodes.

A heavily condensed summary of the endpoint code can be seen in Figure 12. To increase readability, rows concerning matters such as RNG endpoint selection and error handling have been removed from the figure. However, as can be seen when comparing the first few rows of the figure in hand to the figure displaying RNG endpoint's implementation (see Figure 4), the creation of the endpoint classes themselves have several differences. Scala Futures are required to perform HTTP requests with Akka HTTP. However, the usage of Futures requires an *ExecutionContext* (Haller, Prokopec, Miller, Klang, Kuhn, & Jovanovic n.d.). Akka's HTTP client also requires something called an *ActorSystem*, which is an Akka concept. Additionally, unmarshalling requires a *Materializer* which is factory for stream execution engines (Streams Quickstart Guide n.d.). In this case *ActorMaterializer* is used as the *Endpoints* class already supplies it to every endpoint contained within it.

These three concepts were not explored any further in this study, but the requirement for their usage should nevertheless be noted.

```
// ...
class MultiplyEndpoint(
  implicit ec: ExecutionContext,
  mat: ActorMaterializer,
  system: ActorSystem) {
  // ...
  val multiplyRoute: Route = {
    pathPrefix("multiply") {
      (get & path(IntNumber)) { number =>
        // ...
        val rngEndpointFuture: Future[HttpResponse] =
          Http().singleRequest(HttpRequest(uri = rngUri))
        onComplete(rngEndpointFuture) {
          case Success(res) =>
            val resAsString: Future[String] = Unmarshal(res).to[String]
            onComplete(resAsString) {
              case Success(n) => complete((number * n.toInt).toString)
              case Failure(_) => complete(HttpResponse(
                status = StatusCodes.BadRequest,
                entity = "Something went wrong"))
            }
          case Failure(_) => complete(HttpResponse(
            status = StatusCodes.ServiceUnavailable,
            entity = "RNG Service unavailable"))
        }
      }
    }
    // ...
  }
}
```

Figure 12. Summary of Multiplication endpoint

Apart from the primary endpoint itself, everything regarding the handling of the service's endpoints is nearly the same as in RNG Service. This includes binding and handling, the used local IP address, and the use of *Endpoints* class which is imported in the main application itself. However, for the sake of brevity, Multiplication Service's binding and handling is displayed in Figure 13. One of the most visible changes is the inclusion of an additional check in the *HealthCheckRoutes* handling which is addressed in the next chapter. However, from the usage point of view the

most important change is the port that the service uses. 8082 was selected instead of 8080 to avoid port collisions if both services were to be run on the same environment for a reason or another. Running both services on the same machine was never tested in action, as it was not found to be of very important for the success of the study.

```
// ...
object Application extends App with JsonSupport {
  val modules = new AllModules

  import modules._
  // ...
  Http().bindAndHandle(
    HealthCheckRoutes.health(
      multiplyCheck, multiplyCheck2, googleCheck, rngAvailabilityCheck)~
      modules.endpoints.routes, "0.0.0.0", 8082).onComplete {
    case Success(b) => system.log.info(
      s"application is up and running at
${b.localAddress.getHostName}:${b.localAddress.getPort}")
    case Failure(e) => system.log.error(
      s"could not start application: {}", e.getMessage)
  }
}
```

Figure 13. Binding and handling Multiplication Service's endpoints

6.3.3 Health heck endpoint

The way that the health check endpoint is handled in Multiplication Service is nigh identical to RNG Service. As RNG Service, Multiplication service has two checks that verify the availability of the primary endpoint, in this case */multiply/1*. One of the checks (*multiplyCheck*) is created as a plain *asyncHealthCheck* object like in Figure 6, while the other (*multiplyCheck2*) is created using the exact same function as in Figure 7. That function is also used to create the *googleCheck* in Figure 13 which is the same check as in RNG Service. However, in addition to the three previously mentioned checks, the service also implements a fourth check inspecting RNG Service's availability. Structurally *rngAvailabilityCheck* is very similar to the other HTTP endpoint checks using *asyncHealthCheck*. However, instead of making a simple HTTP GET request using Akka HTTP, the check uses Helm to request a list of healthy RNG

service nodes from Consul. If the received list is empty, the check fails, and the Multiplication Service is marked as unhealthy. As with RNG Service, the endpoint is available at the service's */health* subdirectory and requires the GET parameter *full* to be set to *true* to show complete results. Binding and handling of the health checks is displayed in Figure 13.

Like RNG Service, in addition to the health check endpoint implemented with *healthchecks* library, Multiplication Service also implements the redundant health check endpoint included in the service template. For all intents and purposes the endpoint is the same as in RNG Service apart from the targeted address. However, the order of operations is marginally different, and one additional logging statement was added.

6.3.4 Consul integration

The Multiplication service does not use Helm to register itself during the service's start up. Instead the service and its health checks are registered using Consul configurations. In addition to the Consul agent's configuration, two additional configuration files are created; one is for the Multiplication service and its checks, and the other is for checks run against an external service registered upon Multiplication service's start up. As with the server nodes, all the configuration files are generated and put in place by the Vagranfile used to deploy the machines for the proof of concept environment. Thus, the service and check registrations happen alongside agent configuration as the Consul agent starts, as opposed to registration upon the service's startup.

The Consul agent configuration is quite like the one used for RNG Service and can be found in Figure 14. The key difference to RNG Service is the inclusions of *recursors*, *addresses*, and *ui* fields. Giving the recursors field a value corresponding to a DNS server enables Consul to resolve DNS queries for addresses outside of Consul's service domain. The setting is added in preparation for external services. The *addresses* field dictates which interfaces can communicate with Consul. When the field is omitted from the configuration, the address defaults to "127.0.0.1", thus allowing only local loopback connections to communicate with Consul. The other address corresponds to the network interface accessible by the machine hosting the

virtual machine running the Consul agent. This allows the host machine to connect to Consul's HTTP API. When combined with the *ui* parameter which enables Consul HTTP API's graphical user interface, service's, heath checks, and other matters registered to Consul can be viewed and modified from the host machine. In this study the UI is used for demonstrative purpose, and the additional address is needed for its viewing as the machines in the Consul cluster were not configured to have a graphical interface. (Configuration n.d.)

```
{
  "data_dir": "/tmp/consul",
  "server": false,
  "node_name": "client-01",
  "bind_addr": "172.20.20.10",
  "retry_join": [
    "172.20.20.20"
  ],
  "addresses": {
    "http": "127.0.0.1 10.0.2.15"
  },
  "ui": true,
  "recursors": [
    "8.8.8.8"
  ]
}
```

Figure 14. Multiplication Service's Consul agent configuration

The Consul service definition for the Multiplication service is contained within a separate configuration file along with its health check definitions. As can be seen from Figure 15, the service is registered to Consul with the name "mystery-multiplier" as it was the service's initial project name. The checks seen in the configuration file are simple HTTP GET checks which only take in account the HTTP status code of the response and are indifferent to the content returned by the request.

```
{
  "service": {
    "name": "mystery-multiplier",
    "port": 8082,
    "checks": [
      {
        "name": "Mystery multiplier health endpoint",
        "http": "http://localhost:8082/health?full=true",
        "interval": "15s"
      },
      {
        "name": "Multiplication endpoint",
        "http": "http://localhost:8082/multiply/2",
        "interval": "15s"
      }
    ]
  }
}
```

Figure 15. Multiplication Service's Consul configuration

6.3.5 External service registration and Time to Live health check

The third configuration file mentioned in the previous chapter is reserved for a time-to-live check for a registered external service. The time-to-live check is run using a scheduled Akka actor in the Multiplication service itself and will be elaborated on shortly. The check definition is quite simple, and the most significant parameter within it from the point of view of this study is the *ttl* duration. As discussed in chapter 3.4.4, Consul checks can be implemented using multiple different methods, and assigning the *ttl* parameter defines the check as a Time to Live check. It also defines the time window in which the check status must be updated. As shown in Figure 16, in this case the time to live expiration window is 90 seconds. (Checks n.d.)

```
{
  "check": {
    "id": "google-check",
    "name": "Google frontpage headers",
    "notes": "Mystery Multiplier should update this every 60 seconds",
    "ttl": "90s"
  }
}
```

Figure 16. Time to Live health check

Google search was registered as an external service for Consul with the name "search". Since Helm lacks support for registering external services, the registration is done directly via Consul's HTTP API using Akka's HTTP client upon Multiplier Service's start-up. The data used for the service registration is based on Consul's *Registering an External Service* guide with adaptations made to accommodate HTTPS (port 443 used instead of 80) and later addressed issues with strict Helm queries. The JSON containing the service definition were built using case classes and Akka HTTP's included *SprayJsonSupport* using the *DefaultJsonProtocol* of the dedicated *spary-json* library as depicted in Figure 17. As *RootJsonFormat* was used for *JsonSupport*'s formats, the case classes can be marshalled into valid JSON objects in the main *Application* object by using the *JsonSupport* trait. The JSON resulting from the marshalling could then be sent to local Consul agent's Catalog registration HTTP endpoint (http://127.0.0.1:8500/v1/catalog/register) as a PUT payload which then registers the service. (Registering an External Service n.d.)

```
// ...
// https://doc.akka.io/docs/akka-http/current/common/json-support.html
case class MinimalService(Service: String, Port: Int)
case class TaggedAddresses(lan: String, wan: String)
case class ExternalService(
  Datacenter: String, Node: String, Address: String,
  Service: MinimalService,
  NodeMeta: Map[String, String] = Map("external-owner"->"google"),
  TaggedAddresses: TaggedAddresses
)
trait JsonSupport extends SprayJsonSupport with DefaultJsonProtocol {
  implicit val minimalServiceFormat: RootJsonFormat[MinimalService] =
    jsonFormat2(MinimalService)
  // Repeat previous for each case class
  // ...
}
// ...
object Application extends App with JsonSupport {
  // ...
}
```

Figure 17. Defining case classes for external service

The execution of the Time to Live check is done in the service's main *Application* object. It uses a purpose-built Akka actor combined with an Akka scheduler which

sends messages to the actor at regular intervals. While the actor created to update the check's state has more lines of code than might be necessary (see source codes behind Appendix 1), its structure is quite simple. When *TTLActor* receives an "update" message, it first fetches a list of external "search" service instances from Consul via Helm. If none are found, the check's status is updated to "failure" based on the check's ID. Prior to adding *NodeMeta* and *TaggedAddresses* fields to the external service definition, the fetching would fail due to Helm's strict expectations. If any "search" instances are found, a simple HTTP HEAD request is sent to the address and the received HTTP status is inspected. If the status indicates success the check's status is updated to "passing", otherwise it is changed to "warning". Akka's HTTP client is used to update the check state as Helm currently lacks support for Time to Live checks.

After the actor definition was done, it still needed to be combined with a scheduler to ensure that the actor received an "update" message at regular intervals. Since the scheduling logic itself did not need to be very complex, a slightly modified version of Akka's scheduler example was used (Scheduler n.d.). The resulting scheduler sends an "update" message to an instance of *TTLActor* every 60 seconds, waiting 30 seconds after the service's start-up before sending the first message. The initialization of the actor and the timer is displayed in Figure 18. Both objects were initialized in the *Application* object right before binding and handling the service's endpoints.

```
// ...
  val ttlActor = system.actorOf(Props(new TTLActor("google-check")))
  val cancellable =
    system.scheduler.schedule(
      30.seconds, 60.seconds, ttlActor,"update"
    )
// ...
```

Figure 18. Time to Live health check with Akka scheduler

It is worth noting that this is not the recommended way to health check external Consul services. Instead, HashiCorp advices using a separate Consul ESM companion daemon to monitor the health of external services. One of the issues with checking

an external service with a local check is that the check is not properly associated with the service it is supposed to be checking on. Since the external service is registered directly to Consul's *Catalog*, which is the registry for every check, service, and node within Consul, when the application starts, instead of registering it to a certain Consul Agent, the service is not directly associated with any node. As the association between a separate health check and a service is done using node-specific service IDs, the check cannot be properly associated with the external service's ID. Thus, the check is instead registered to the node, which in this case is an unwanted side effect. There might be ways to work around this issue; however, that would require a dramatic overhaul to the way the external service registration and health checking which would be unattainable at this point of the study. Nevertheless, registration of external services and Consul's time-to-live health checks are demonstrated by this set up, which is why this piece of implementation was not completely removed from the final proof-of-concept. (Checks n.d.; Koehler 2018; Registering an External Service n.d.)

6.3.6 Running the service

Deploying the Multiplication Service is very similar to RNG Service. Assuming that the Consul server cluster and at least one of the RNG Service instances have already been set up, after the used Vagrantfile has provisioned the virtual machine dedicated for Multiplication Service, two Vagrant SSH instances are used to establish connections to the machine. The lack Consul server cluster and RNG Service instance do not prevent the service from starting; however, having both in a working order is nevertheless recommended for the proper function of the service.

As with RNG Service, the first connection is used to start the Consul Agent with the command *consul agent -config-dir=/etc/consul.d* which loads up Consul's configuration files and starts its client Agent if the machine was properly provisioned. The second shell is used to navigate to */home/vagrant/mystery-multiplier*. In there, the command *sbt* is run to open the Sbt shell. Like with RNG Service, the Sbt shell command *reStart* is then used to compile and start the service. Both SSH connections should be left open, as logging out of the SSH shell ends the tasks running within that shell.

After starting the service, its multiplication endpoint can be found from the path */multiply/x* where *x* can be replaced with any integer. As the service is bound to listen to all of the machines' network interfaces (see chapter 6.2.2 and Figure 5), when combined with a VirtualBox port forwarding rule forwarding all TCP traffic in port 8082, Multiplication service is accessible from the host machine as well. The primary endpoint can be found from http://127.0.0.1:8082/multiply/. Additionally, since the Consul agent configuration on the machine enables Consul's UI, another TCP port forwarding rule was made for port 8500. This makes the Consul UI accessible from the host machine as well with the address http://127.0.0.1:8500/ui.

6.4 Deployment

The whole proof of concept environment can be deployed by running *vagrant up* in the *Vagrant* folder of the project repository (see Appendix 1). The Vagrantfile contained in the folder is configured to start and set up all the virtual machines required for demonstrating the proof of concept. The setting up includes setting up the machines' private network interface, port forwarding rules, installing dependencies such as OpenJDK and Consul, generating all static Consul configuration files, and copying the services' source codes to the appropriate machines. Each of the Consul nodes is configured to join the first server node with its static private IP address. This is somewhat antithetical to how Consul's clustering and service discovery works, as the whole point of service discovery revolves around not hard-coding target IPs. However, based on quick Ad Hoc testing, it was found that joining to any of the nodes already in the cluster works, even if that node is not a server node.

After all the machines are started, each one of them is connected to with one or more SSH connections established with *vagrant ssh*. Each Consul server node (s1, s2, s3) is connected to with a single SSH connection each. Those SSH connections are then used to start the Consul agent on each of the Server nodes with the command *consul agent -config-dir=/etc/consul.d*. The order in which the agents are started has little to no effect on the overall usage of the server cluster, but to minimize the count of warning and error messages in Consul's output, it is recommended that Consul

agent is first started on the machine *s1*. The designed start-up order for the machines along with their purpose are listed in Table 3.

Table 3. Virtual Machines and start-up order

Name	Private address	Purposes	SSH shells required
s1	172.20.20.20	"Main" Consul server node, primary joining targer	1
s2	172.20.20.21	Consul server node	1
s3	172.20.20.22	Consul server node	1
c2	172.20.20.11	Consul client node, RNG Service host	2
c3	172.20.20.12	Consul client node, RNG Service host	2
c1	172.20.20.10	Consul client node, Multiplication Service host, Consul UI host	2

The Consul agent nodes (c1, c2, c3) are require two SSH connections each for proper operation. The first connection is used to start Consul agent the same way it is started with the server nodes. After that, the second connection is used to start the service's themselves. To run the service, Sbt shell is opened in the directory containing the service's source codes, that being either */home/vagrant/multiply-proto* or */home/vagrant/rng-proto* depending on which service is on the machine. The command *reStart* is used in Sbt shell to start the service. The services should be starter after the Consul agents to ensure proper functioning of the services. Similarly, Multiplication service on the machine *c1* should be starter only after starting RNG Service on both *c2* and *c3*.

The deployment has been tested using a Windows 10 host with Vagrant version 2.2.4 and Oracle VirtualBox 6.0.4. The start-up has irregular SSH connection issues which require the removal of the failed machine and rerunning *vagrant up*. The issue is caused by a missing IP assignment in a virtual machine's interfaces, which seems to be caused by VirtualBox's network adapter or its configuration (Figure 19). The SSH connection can fail also on later starts of the machines. In such cases simply

rebooting the virtual machine is often enough, but in certain instances shutting it down and restarting it using *vagrant up* instead can be necessary. Unfortunately, no proper fix for the issue was found during the study.

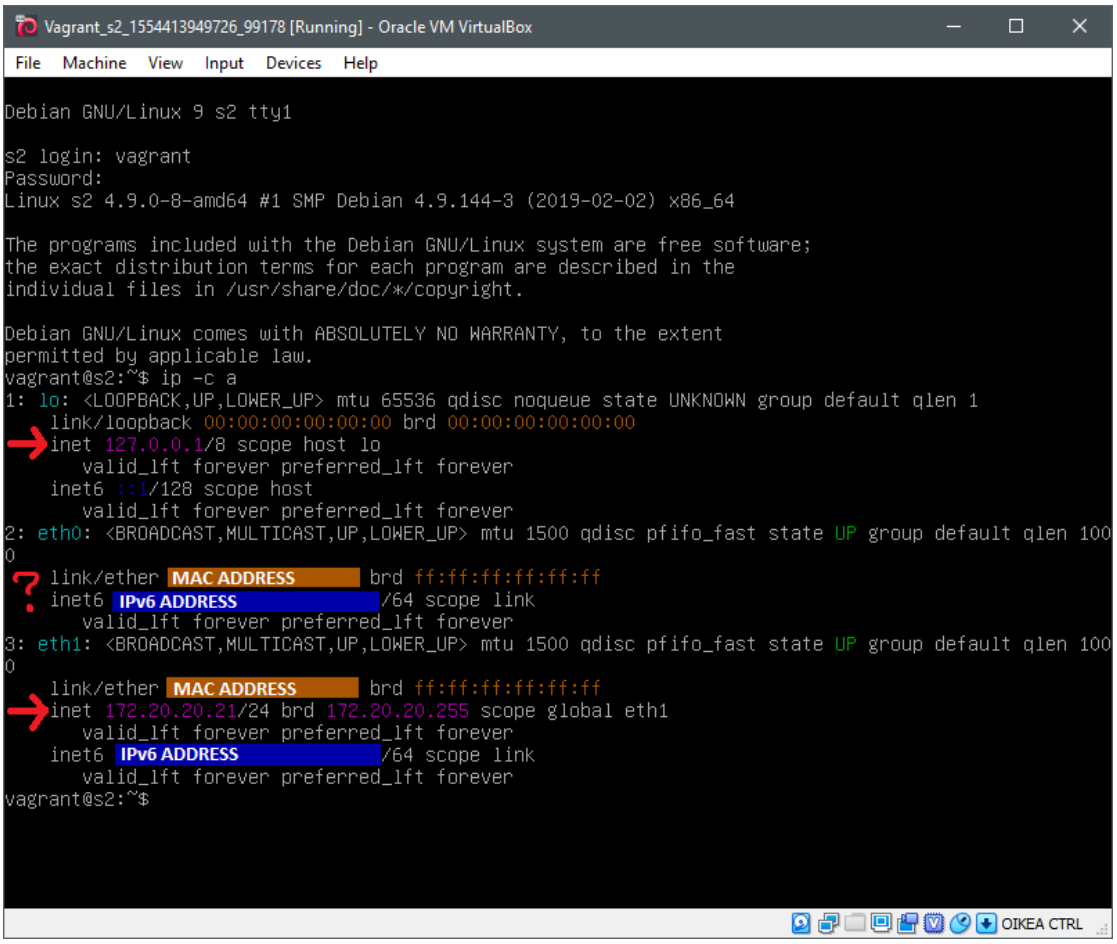

Figure 19. Missing IPv4 address on eth0 (some addresses redacted)

Another issue noticed very late during development is that with certain network issues health checks sending requests to external addresses can go into a state where their respective response futures never complete. The future not completing causes the check itself to never receive a healthy nor an unhealthy status, which then causes the health check endpoint to break. This was noticed when WAN routing was disabled in a local network router connected to the environment's host, causing all Google checks to never complete. However, if the host is simply disconnected from the network, the checks fail as expected. This was not tested extensively, and thus is possible that the issue lies more on the router instead of the application itself.

7 Results

The implementation resulted in a system with a cluster of virtual machines with health checked services connecting with each other via Consul. It fulfils most of the requirements in chapter 2.2. While some of the stretch-goal requirements are not met by the proof of concept, every primary requirement was fulfilled, mostly without issues. However, the proof of concept should not be used as a direct reference for production as it contains several issues, such as improper usage of externally registered services and their health checks. It nevertheless adequately demonstrates service discovery and health check design patterns.

With a complete proof of concept environment running, the study's results can be displayed and verified. As the proof of concept's Multiplication Service and Consul UI can be accessed from the environment's host machine, most of the verifications can be done from it without having to directly access the virtual machines themselves. The primary target used for verifying the results of the study is Multiplication Service, as it fulfils the largest part of the fulfilled requirements. If the service's multiplication endpoint works as expected, requirements FR-01 and NR-01 are fulfilled by default, as Multiplication Service does not have any preconfigured addresses set up for RNG Service, and it consumes data requested from RNG Service, acting as RNG Service's client.

Multiplication Service's function is displayed in Figure 20 alongside output logs of both RNG Service instances, which correlate with requests sent to Multiplication Service. In this case the requests are sent from the proof of concept environment's host machine which has a port-forwarded connection to *c1* virtual machine (see chapter 6.3.6). The requests are sent with a simple curl command, and echo is only used to add a new line for the console for the sake of brevity. Request with paths beginning with "172.20.20" are from multiplication service, as it interacts with RNG Services through their public IPs. These interactions are underlined in Figure 20. The rest of the requests are caused by the local Consul agent running tests against the service.

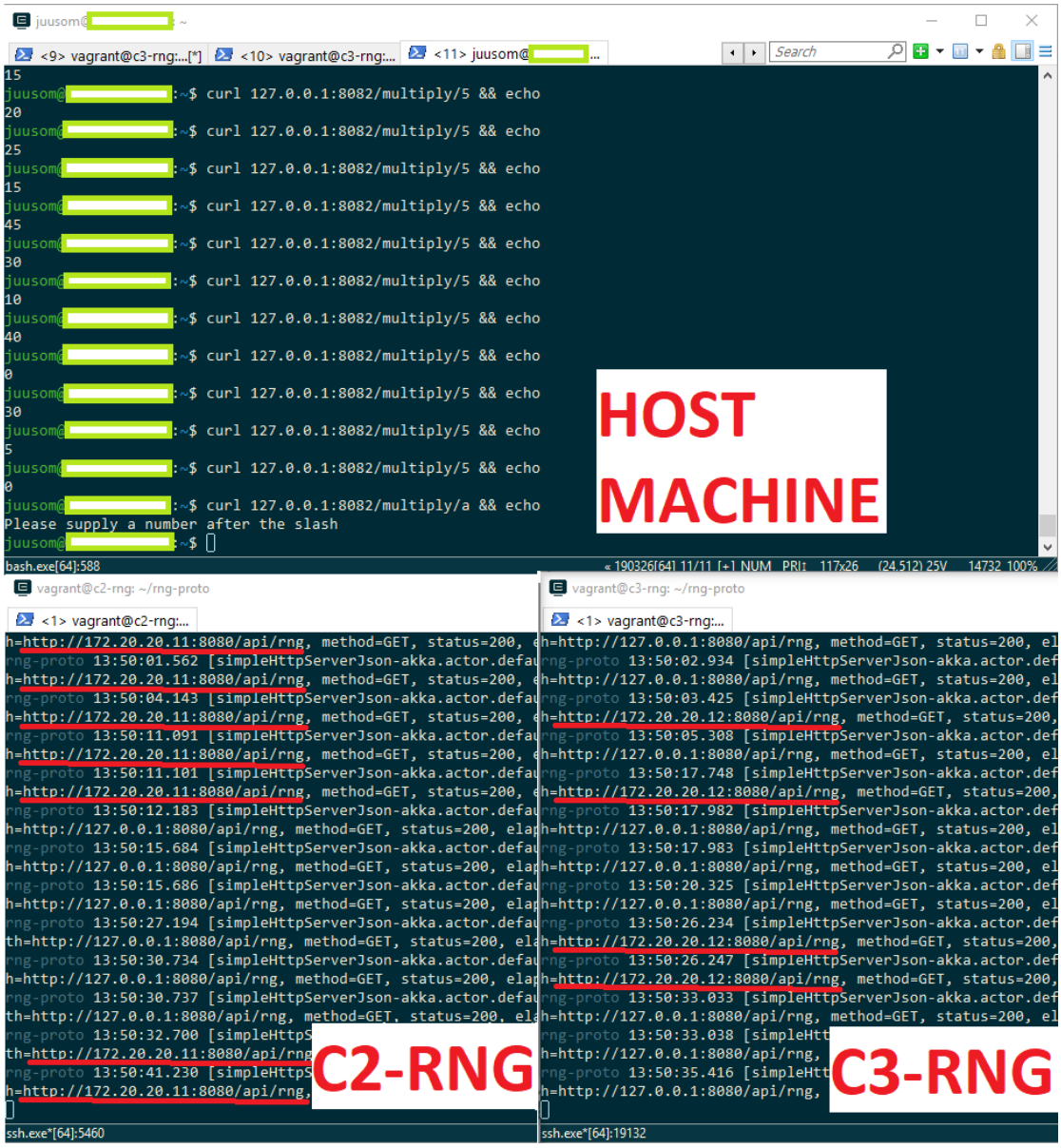

Figure 20. Multiplication service in action

FR-01 can be further verified by stopping the Consul agent on virtual machine *c1*, which renders any RNG Service instances unreachable for Multiplication Service (Figure 21). However, supplying the endpoint with a non-integer value produces the same error regardless of Consul agent's state.

```
juusom@[redacted]:~$ curl 127.0.0.1:8082/multiply/5 && echo
There was an internal server error.
juusom@[redacted]:~$ curl 127.0.0.1:8082/multiply/5 && echo
There was an internal server error.
juusom@[redacted]:~$ curl 127.0.0.1:8082/multiply/a && echo
Please supply a number after the slash
juusom@[redacted]:~$
```

HOST MACHINE

Figure 21. Multiplication without Consul agent

Navigating to Multiplication service's */health* endpoint provides the information required for the verification of requirement FR-02 (Figure 22). Unfortunately, no direct method was implemented for toggling the status of specific health checks. However, the same effect could be simulated by disconnecting the whole environment from the internet.

127.0.0.1:8082/health?full=true

127.0.0.1:8082/health?full=true


```
// 20190415215115
// http://127.0.0.1:8082/health?full=true

{
  "status": "healthy",
  "check_results": [
    {
      "name": "multiply health check",
      "severity": "Fatal",
      "status": "healthy",
      "messages": [

      ]
    },
    {
      "name": "multiply check 2",
      "severity": "Fatal",
      "status": "healthy",
      "messages": [

      ]
    },
    {
      "name": "google check",
      "severity": "NonFatal",
      "status": "healthy",
      "messages": [

      ]
    },
    {
      "name": "check rng endpoint availability",
      "severity": "Fatal",
      "status": "healthy",
      "messages": [

      ]
    }
  ]
}
```


Figure 22. Multiplication service's health endpoint

By disconnecting the host machine from the internet, every Google check in any service can be seen changing its state, as is displayed in Figure 23. The same figure also shows how Multiplication Service to turns unhealthy when all RNG Service instances fail, though this behaviour can be better replicated by simply stopping both RNG Service instances instead. However, it was also noticed that network disruption

can cause the health checks to act erratically. Instead of returning a JSON with the health check statuses, the health endpoint takes multiple seconds to load and eventually returning an HTTP status code 503. This marks the endpoint's service instance (in this case RNG Service) as unhealthy to Consul. While this was not extensively tested, it is suspected that this is caused by misconfigured timeouts in checks requesting Google's front page, as they are the only checks using resources outside of the test network.

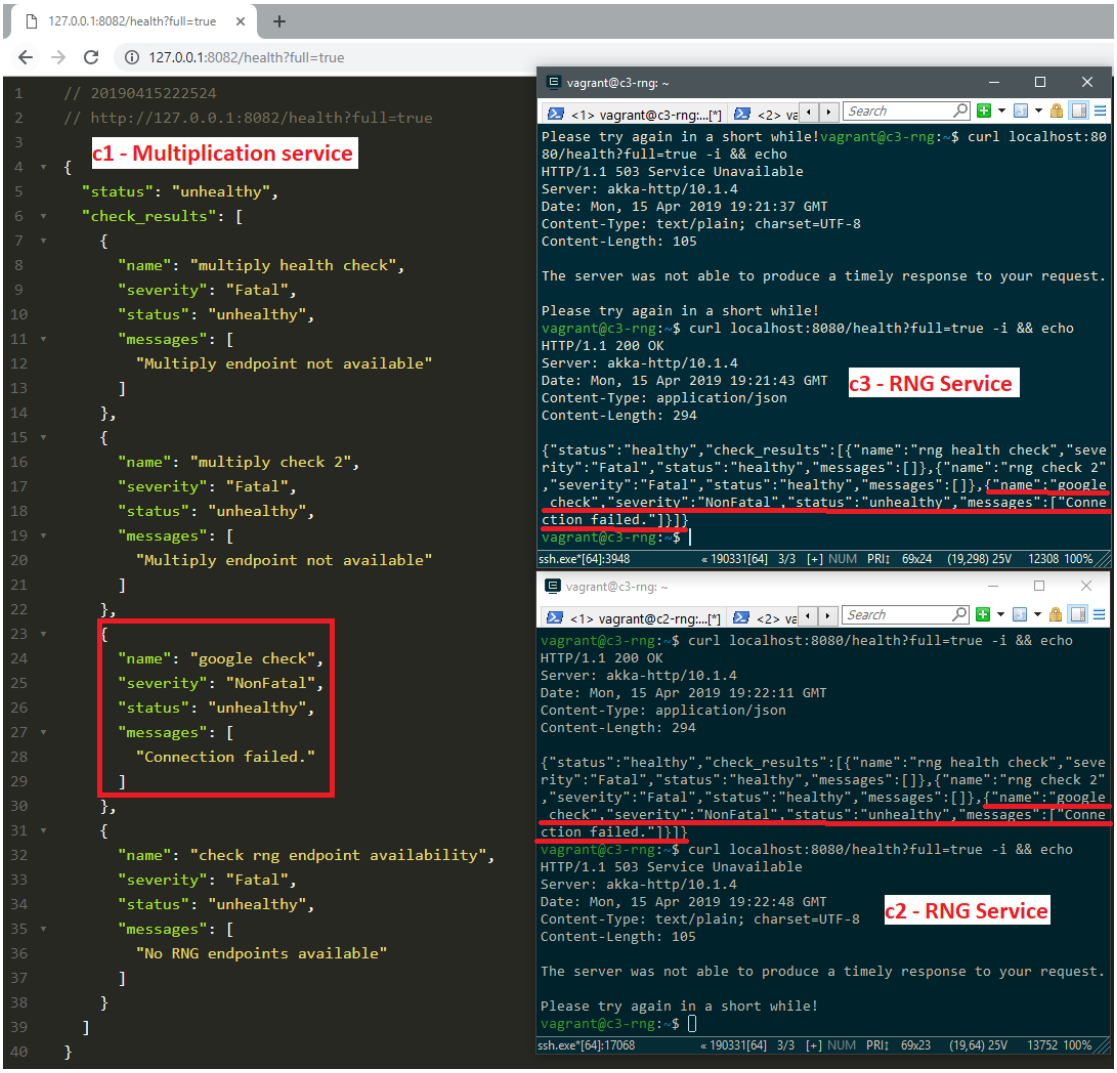

Figure 23. Network disruption and failing Google checks

Verifying FR-03 requires a small modification to the source code of one of the RNG Service instances. In this case the instance running on virtual machine *c3* is modified. Changing Google check's target to an invalid URI (Figure 24) causes the check to

consistently fail and modifying the check severity to “Fatal” makes sure that the service instance is marked as unhealthy by Consul.

```
vagrant@c3-rng: ~
<1> vagrant@c3-rng:...[*]   <2> vagrant@c3-rng:...[*]   <3> vagrant@c3-rng:...
      response =>
      if (response.status.isSuccess) healthy else unhealthy("RNG endpoint not available")
    }
  }

  // Does not really server a purpose?
  val rngCheck2 = httpHealthCheck("rng check 2", "http://127.0.0.1:8080/api/rng")
  val googleCheck = httpHealthCheck(target = "https://invalid.address", name = "google check", severity = Severity.Fatal)

  // Consul stuff
  // TODO: Check if runSync can be used instead
  val client = Http1Client[IO]().unsafeRunSync()
  val baseUrl = uri("http://127.0.0.1:8500")

  val interpreter = new Http4sConsulClient(baseUrl, client)
:
119 chars {3,8}-{121,8}:{3,8} stream selection   « 190331[64]  3/3  [+] NUM  PRI  126x16  (121,8) 50V  12308 100%
```

Figure 24. Breaking the Google check

After recompiling and restarting the service the check starts failing as expected (Figure 25) and thus the service instance is marked as unhealthy by Consul. After Consul notices that the instance is unhealthy, it no longer receives requests from Multiplication Service. This can be verified by requesting Multiplication endpoint and comparing the output logs of the RNG Service instances. As can be seen in Figure 26, RNG Service on machine *c2* receives a disproportionate number of requests from its public interface “172.20.20.11”, while the modified *c3* receives no requests from its equivalent interface.

```
vagrant@c3-rng:~$ curl http://127.0.0.1:8080/health?full=true -s | jq
{
  "status": "unhealthy",
  "check_results": [
    {
      "name": "rng health check",
      "severity": "Fatal",
      "status": "healthy",
      "messages": []
    },
    {
      "name": "rng check 2",
      "severity": "Fatal",
      "status": "healthy",
      "messages": []
    },
    {
      "name": "google check",
      "severity": "Fatal",
      "status": "unhealthy",
      "messages": [
        "Connection failed."
      ]
    }
  ]
}
vagrant@c3-rng:~$
```

Figure 25. Results of the malformed Google check

```
C3-RNG
=http://127.0.0.1:8080/api/rng, method=GET, status=200, elapsed
rng-proto 20:43:11.926 [simpleHttpServerJson-akka.actor.default
=http://127.0.0.1:8080/api/rng, method=GET, status=200, elapsed
rng-proto 20:43:11.927 [simpleHttpServerJson-akka.actor.default-dispatcher-8]
=http://127.0.0.1:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:43:14.502 [simpleHttpServerJson-akka.actor.default-dispatcher-18]
h=http://127.0.0.1:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:43:26.972 [simpleHttpServerJson-akka.actor.default-dispatcher-18]
h=http://127.0.0.1:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:43:26.972 [simpleHttpServerJson-akka.actor.default-dispatcher-18]
h=http://127.0.0.1:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:43:29.506 [simpleHttpServerJson-akka.actor.default-dispatcher-3]
=http://127.0.0.1:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:43:42.044 [simpleHttpServerJson-akka.actor.default-dispatcher-2]
=http://127.0.0.1:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:43:42.044 [simpleHttpServerJson-akka.actor.default-dispatcher-2]
=http://127.0.0.1:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:43:44.510 [simpleHttpServerJson-akka.actor.default-dispatcher-6]
=http://127.0.0.1:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:43:57.082 [simpleHttpServerJson-akka.actor.default-dispatcher-17]
h=http://127.0.0.1:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:43:57.082 [simpleHttpServerJson-akka.actor.default-dispatcher-17]
h=http://127.0.0.1:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:43:59.515 [simpleHttpServerJson-akka.actor.default-dispatcher-18]
h=http://127.0.0.1:8080/api/rng, method=GET, status=200, elapsedTime=0
```

```
C2-RNG
=http://172.20.20.11:8080/api/rng, method=GET, status=200, el
rng-proto 20:44:04.017 [simpleHttpServerJson-akka.actor.defau 2]
=http://172.20.20.11:8080/api/rng, method=GET, status=200, el
rng-proto 20:44:04.342 [simpleHttpServerJson-akka.actor.default-dispatcher-15]
h=http://172.20.20.11:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:44:04.644 [simpleHttpServerJson-akka.actor.default-dispatcher-5]
=http://172.20.20.11:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:44:04.972 [simpleHttpServerJson-akka.actor.default-dispatcher-2]
=http://172.20.20.11:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:44:05.275 [simpleHttpServerJson-akka.actor.default-dispatcher-2]
=http://172.20.20.11:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:44:05.554 [simpleHttpServerJson-akka.actor.default-dispatcher-13]
h=http://172.20.20.11:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:44:05.873 [simpleHttpServerJson-akka.actor.default-dispatcher-13]
h=http://172.20.20.11:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:44:06.206 [simpleHttpServerJson-akka.actor.default-dispatcher-3]
=http://172.20.20.11:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:44:06.685 [simpleHttpServerJson-akka.actor.default-dispatcher-2]
=http://172.20.20.11:8080/api/rng, method=GET, status=200, elapsedTime=1
rng-proto 20:44:07.713 [simpleHttpServerJson-akka.actor.default-dispatcher-3]
=http://172.20.20.11:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:44:08.500 [simpleHttpServerJson-akka.actor.default-dispatcher-15]
h=http://172.20.20.11:8080/api/rng, method=GET, status=200, elapsedTime=0
rng-proto 20:44:08.500 [simpleHttpServerJson-akka.actor.default-dispatcher-15]
h=http://172.20.20.11:8080/api/rng, method=GET, status=200, elapsedTime=0
```

Figure 26. Discrepency between requests to *c3* and *c2*

The service's Consul health status can also be seen from Consul's UI (Figure 27). Consuls UI can also be used to check other services' and nodes' health, as well as to manipulate Consul's key/value storage, ACLs and intentions which fall outside of the study's scope.

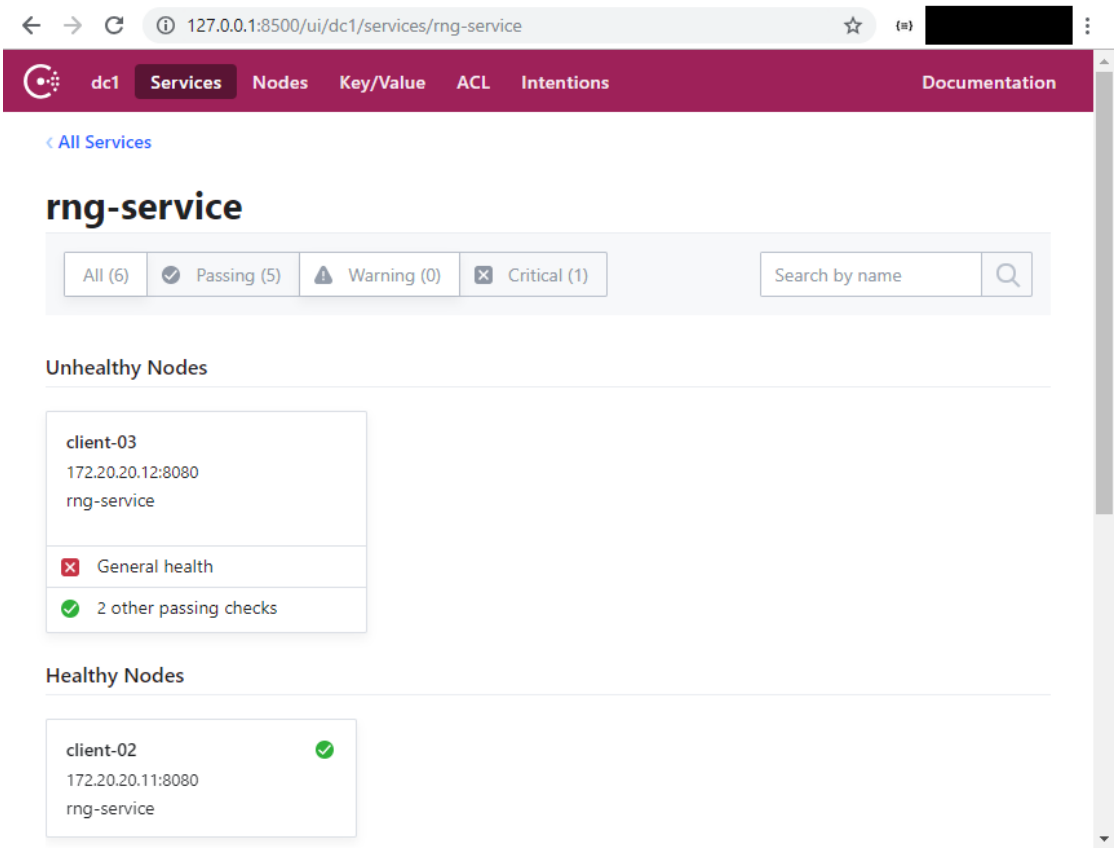

Figure 27. Consul UI

The resulting system also fulfils some of the stretch goal requirements defined in chapter 2.2. Feature FR-04 is mostly implementation-side (see chapter 6.3.5) and not very visible from the service itself. However, the external service is accompanied with a Consul Time to Live health check which fulfils requirement FR-05. While the check is unfortunately not properly associated with the external service itself in Consul's registry, it nevertheless demonstrates a working Time to Live health check. This is displayed in Figure 28. It displays the failed Time to Live check and the output of the last time it passed.

```
// 20190415173434
// http://127.0.0.1:8500/v1/health/node/client-01

[
  {
    "Node": "client-01",
    "CheckID": "google-check",
    "Name": "Google frontpage headers",
    "Status": "passing",
    "Notes": "Mystery Multiplier should update this every 60 seconds",
    "Output": "List(Date: Mon, 15 Apr 2019 14:31:42 GMT, Expires: Wed, 01 Jan 1800 00:00:00
GMT, Cache-Control: private, max-age=0, P3P: CP=\"This is not a P3P policy! See
g.co/p3phelp for more info.\", Server: gws, X-XSS-Protection: 0, X-Frame-Options:
SAMEORIGIN, Set-Cookie: 1P_JAR=2019-04-15-14; Expires=Wed, 15 May 2019 14:31:42 GMT;
Domain=google.com; Path=/, Set-Cookie: NID=181=Bm7EmS5K9Y71JfxfewxJ9wL-
wvEgosa59CRugBvHeh_jpYOHq6TEyiedeqxKNwfo1qvJfapHMq_TUMDc3OAgOjwSEcc33ZeprAJ7SiMsTJ9bsKFOlqJ
Ov-_JsdAwXv1TSY0d3yKxGAN_U6dQ_YemsMBsEOfrT6D7tiiPLt6AMAo; Expires=Tue, 15 Oct 2019 14:31:42
GMT; Domain=google.com; Path=/; HttpOnly, Alt-Svc: quic=\":443\"; ma=2592000;
v=\"46,44,43,39\", Accept-Ranges: none, Vary: Accept-Encoding)",
    "ServiceID": "",
    "ServiceName": "",
    "ServiceTags": [

    ],
    "Definition": {

    },
    "CreateIndex": 1598,
    "ModifyIndex": 1601
  },
  {
    "Node": "client-01",
    "CheckID": "serfHealth",
    "Name": "Serf Health Status",
    "Status": "passing",
    "Notes": "",
    "Output": "Agent alive and reachable",
    "ServiceID": ""
```


Figure 28. Passing Google Time to Live health check

The technical requirement NR-02 is nigh impossible do display from a user's point of view. However, it can be seen fulfilled in chapter 6, as both services were created with Scala language. It could be argued that the requirement is not completely fulfilled from Akka point of view, since Helm utilizes Http4s instead of Akka HTTP for its requests. Nevertheless, its usage was strictly limited to required interactions with Helm and is thus not in conflict with Akka.

8 Conclusions

To summarize the results of the study: it was found that both service discovery and health checks can be included in a microservice system built using the technologies

required by the study. Most of the study's requirements were met during the implementation of the proof of concept system, and the unfulfilled requirements had been categorized as non-mandatory stretch goals prior to the implementation. As most of the requirements were met, it can be argued that most of the tools used in the implementation are likely suitable for production usage as well. Consul itself with its robust set of features seems to be quite suitable for production level usage, and as so is a good candidate for a service discovery. Everpeace's *healthchecks* library seems suitable for larger scale usage as well, and there is not much to complain about it. However, Helm was found to have plenty of room for improvement, as in its current form it is not fully compatible with new Consul versions. This could be remedied to a degree by copying of the current Helm codebase and making an improved version of it as the library is released under a MIT license.

The potentials for synergy between service discovery and health checks were noticeable in the proof of concept implementation as was seen with the verification of requirement FR-03. By making sure that only healthy service instances are available in the service registry, situations where a service gets connected to a malfunctioning instance of a dependency service can be reduced, given that the health checks themselves are defined in a way that makes them properly indicative of the health of the service instance. Additionally, the fact that using services with Consul only requires one-time configuration for joining the cluster and finding instances of dependency services reduces the need for configuration. This is demonstrated by Multiplication Service not using pre-defined addresses for either of the RNG Endpoints. While the study was done in a relatively small scale, this advantage should nevertheless translate to larger projects as well.

While the proof of concept displays the ideas of service discovery, it does so with a low count of services. Consul's HTTP API is perfectly usable for service discovery purposes in the scope of the study, but the way that it is utilized in the study might not be optimal in every scenario. Unfortunately, Consul's DNS API was not explored at all during the study. Consul's service mesh feature called Connect might be a better fit for production than how service discovery is done in this thesis. Additionally, if the underlying mechanics of Connect are similar enough to the ones in other service mesh solutions, it should be possible to use the service with multiple

different kind of service proxies or underlying service mesh systems without any major changes to the service itself. This might allow for looser coupling between Consul and the service which can be advantageous if the whole service system must be adaptable to multiple different kinds of environments.

In hindsight it can be said with confidence that the scope for the study was wider than necessary. The unfortunate outcome of the large scope is that neither of the study's subjects was research as deeply as was initially hoped. Either of the subjects would have been more than enough for a full study. There is still room for more research to be done on both of the study's subjects. For example, whether it is feasible to try to pair health checks with some sort of "triggers" that start automatic maintenance tasks in an attempt recover the unhealthy service was not explored at all, though such a system might be of high value with mission critical systems. Additionally, as service mesh solutions seem to be becoming more and more common, exploring Consul's Connect feature as well as other service mesh tools such as Istio and Linkerd 2.x might be worthwhile before rushing to a large-scale implementation. Nevertheless, the study's results are still satisfactory.

References

About Landis+Gyr. N.d. Page on Landis+Gyr's web site. Accessed on 7 April 2019. Retrieved from https://www.landisgyr.com/about/

Adding & Removing Servers. N.d. Consul guide. Accessed on 14 April 2019. Retrieved from https://learn.hashicorp.com/consul/day-2-operations/advanced-operations/servers

Akka Quickstart with Scala. N.d. Page on Akka's developer guide. Accessed on 10 April 2019. Retrieved from https://developer.lightbend.com/guides/akka-quickstart-scala/

Arjis, P. 2018. *Monitoring microservices with health checks*. Article on Jaxenter. Accessed on 7 April 2019. Retrieved from https://jaxenter.com/monitoring-microservices-health-checks-142305.html

Baker, R. 2018. *Helm*. Readme-file of Helm's GitHub repository. Accessed on 14 April 2019. Retrieved from https://github.com/Verizon/helm

Basic Provider Usage. N.d. Vagrant documentation. Accessed on 14 April 2019. Retrieved from https://www.vagrantup.com/docs/providers/basic_usage.html

Bootstrapping a Datacenter. N.d. Consul documentation page. Accessed on 14 April 2019. Retrieved from https://www.consul.io/docs/install/bootstrapping.html

Build definition. N.d. Sbt documentation page. Accessed on 14 April 2019. Retrieved from https://www.scala-sbt.org/1.x/docs/Basic-Def.html

Catalog HTTP API. N.d. Consul documentation page. Accessed on 14 April 2019. Retrieved from https://www.consul.io/api/catalog.html

Checks. N.d. Consul documentation page. Accessed on 7 April 2019. Retrieved from https://www.consul.io/docs/agent/checks.html

Client-server architecture. N.d. Article on Encyclopedia Britannica. Accessed on 10 April 2019. Retrieved from https://www.britannica.com/technology/client-server-architecture

Client/Server Architecture. N.d. Dictionary definition on Technopedia. Accessed on 10 April 2019. Retrieved from https://www.techopedia.com/definition/438/clientserver-architecture

Configuration. N.d. Consul documentation page. Accessed on 14 April 2019. Retrieved from https://www.consul.io/docs/agent/options.html

Connect. N.d. Consul documentation page. Accessed on 14 April 2019. Retrieved from https://www.consul.io/docs/connect/index.html

Connect-Native App Integration. N.d. Consul documentation page. Accessed on 12 April 2019. Retrieved from https://www.consul.io/docs/connect/native.html

Consensus Protocol. N.d. Consul documentation page. Accessed on 14 April 2019. Retrieved from https://www.consul.io/docs/internals/consensus.html

Consul Architecture. N.d. Consul documentation page. Accessed on 14 April 2019. Retrieved from https://www.consul.io/docs/internals/architecture.html

Consul KV. N.d. Consul documentation page. Accessed on Accessed on 14 April 2019. Retrieved from https://www.consul.io/docs/agent/kv.html

Dockerfile reference. N.d. Docker documentation. Accessed on 7 April 2019. Retrieved from https://docs.docker.com/engine/reference/builder/#healthcheck

Dadgar, A. 2013. *LICENSE.* License file in Consul's GitHub repository. Accessed on 14 April 2019. Retrieved from https://github.com/hashicorp/consul/blob/master/LICENSE

Ewaschuk, R. 2016. *Monitoring distributed systems*. Except from book *Monitoring distributed systems* hosted as a blog post. Accessed on 13 April 2019. Retrieved from https://www.oreilly.com/ideas/monitoring-distributed-systems

Farcic, V. 2015. *Service Discovery: Zookeeper vs etcd vs Consul*. Entry on Technology Conversations website. Accessed on 12 April 2019. Retrieved from https://technologyconversations.com/2015/09/08/service-discovery-zookeeper-vs-etcd-vs-consul/

Francisco, G. 2017. *akka-http-microservice-templates*. Index of a GitHub repository. Accessed on 14 April 2019. Retrieved from https://github.com/gabfssilva/akka-http-microservice-templates

Freddy. 2018. *Vagrant Consul Demo*. Directory in Consul's GitHub repository. Accessed on 14 April 2019. Retrieved from https://github.com/hashicorp/consul/tree/master/demo/vagrant-cluster

Haller, P., Prokopec, A., Miller, H., Klang, V., Kuhn, R., & Jovanovic, V. N.d. *Futures and Promises*. Scala documentation page. Accessed on 14 April 2019. Retrieved from https://docs.scala-lang.org/overviews/core/futures.html

Hashimoto, M., Coder, F., Piedrafita, M., Rehm, F., Cain, B., & Mohammed, A. 2019. *LICENSE*. License file in Vagrant's GitHub repository with multiple contributors. Accessed on 14 April 2019. Retrieved from https://github.com/hashicorp/vagrant/blob/master/LICENSE

Introduction to Akka. N.d. Page on Akka's documentation site. Accessed on 10 April 2019. Retrieved from https://doc.akka.io/docs/akka/current/guide/introduction.html

Introduction to Consul. N.d. Page on Consul's website. Accessed on 14 April 2019. Retrieved from https://www.consul.io/intro/index.html

IO. N.d. Datatype description on Cats Effect library's documentation. Accessed on 14 April 2019. Retrieved from https://typelevel.org/cats-effect/datatypes/io.htm

Iwaya, A. 2015. *What is the Difference Between 127.0.0.1 and 0.0.0.0?* Article on How-To Geek. Accessed on 14 April 2019. Retrieved from https://www.howtogeek.com/225487/what-is-the-difference-between-127.0.0.1-and-0.0.0.0/

Java Virtual Machine (JVM) & its Architecture. N.d. Online Java tutorial on Guru99. Accessed on 10 April 2019. Retrieved from https://www.guru99.com/java-virtual-machine-jvm.html

Koehler, C. 2018. *Consul and External Services*. Post on HashiCorp's blog. Accessed on 14 April 2019. Retrieved from https://www.hashicorp.com/blog/consul-and-external-services

Lewis, J., & Fowler, M. 2014. *Microservices*. Article on Martin Fowler's blog. Accessed on 9 April 2019. Retrieved from https://martinfowler.com/articles/microservices.html

Library dependencies. N.d. Sbt documentation page. Accessed on 14 April 2019. Retrieved from https://www.scala-sbt.org/1.x/docs/Library-Dependencies.html

Licenses. N.d. List of licensed used by Lightbend. Accessed on 14 April 2019. Retrieved from https://www.lightbend.com/legal/licenses

Maddi, V. 2018. *Productivity at Scale: How We Improved Build Time by 400% at LinkedIn*. Post on LinkedIn Engineering blog. Accessed on 14 April 2019. Retrieved from https://engineering.linkedin.com/blog/2018/07/how-we-improved-build-time-by-400-percent

Miranda, G. 2016. *Which Service Mesh Should I Use?* Sponsored article on The New Stack. Accessed on 12 April 2019. Retrieved from https://thenewstack.io/which-service-mesh-should-i-use/

Morgan, W. 2017. *What's a service mesh? And why do I need one?* Entry on Buoyant' blog. Accessed on 9 April 2019. Retrieved from https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/

Newman, S. 2015. *Building microservices*. Sebastopol: O'Reilly Media.

Omura, S., & Kato, J. 2017. *healthchecks*. Software library's readme-file. Accessed on 7 April 2019. Retrieved from https://github.com/everpeace/healthchecks/blob/master/README.md

Oracle VM VirtualBox. N.d. VirtualBox's home page. Accessed on 14 April 2019. Retrieved from https://www.virtualbox.org/

Providers. N.d. Vagrant documentation. Accessed on 14 April 2019. Retrieved from https://www.vagrantup.com/docs/providers/

Provisioning. N.d. Vagrant documentation. Accessed on 14 April 2019. Retrieved from https://www.vagrantup.com/docs/provisioning/

Reference Model for Service Oriented Architecture 1.0. 2006. OASIS (Organization for the Advancement of Structured Information Standards) Standard. Accessed on 10 April 2019. Retrieved from http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

Registrator. N.d. Registrator documentation page. Accessed on 14 April 2019. Retrieved from http://gliderlabs.github.io/registrator/latest/

Registering an External Service. N.d. Guide on Consul's web site. Accessed on 14 April 2019. Retrieved from https://www.consul.io/docs/guides/external.html

Richardson, C. 2015. *Service Discovery in a Microservices Architecture*. Entry on NGINX's tech blog. Accessed on 8 April 2019. Retrieved from https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

Richardson, C. 2018a. *Pattern: Client-side service discovery*. Article on Microservices.io. Accessed on 12 April 2019. Retrieved from https://microservices.io/patterns/client-side-discovery.html

Richardson, C. 2018b. *Pattern: Health Check API*. Article on Microservices.io. Accessed on 13 April 2019. Retrieved from https://microservices.io/patterns/observability/health-check-api.html

Richardson, C. 2018c. *Pattern: Microservice Architecture*. Article on Microservices.io. Accessed on 12 April 2019. Retrieved from https://microservices.io/patterns/microservices.html

Richardson, C. 2018d. *Pattern: Server-side service discovery*. Article on Microservices.io. Accessed on 12 April 2019. Retrieved from https://microservices.io/patterns/server-side-discovery.html

Richardson, C. 2018e. *Pattern: Service registry*. Article on Microservices.io. Accessed on 12 April 2019. Retrieved from https://microservices.io/patterns/service-registry.html

Rudolph, J., Mathias, Bo, Y., Rogach, Engelen, A., ReadMeCritic, Regadas, F., & Aleksandrov, D. 2017. README.md. Readme-file in Sbt-revolver's GitHub repository. Accessed on 14 April 2019. Retrieved from https://github.com/spray/sbt-revolver/blob/master/README.md

sbt Reference Manual. N.d. Sbt documentation page. Accessed on 14 April 2019. Retrieved from https://www.scala-sbt.org/1.x/docs/index.html

Scala Native. N.d. Page on Scala Native's documentation site. Accessed on 10 April 2019. Retrieved from https://scala-native.readthedocs.io/en/v0.3.8/index.html

Scheduler. N.d. Akka documentation. Accessed on 14 April 2019. Retrieved from https://doc.akka.io/docs/akka/current/scheduler.html

Services. N.d. Consul documentation. Accessed on 14 April 2019. Retrieved from https://www.consul.io/docs/agent/services.html

Service segmentation made easy. N.d. Consul use case page. Accessed on 14 April 2019. Retrieved from https://www.consul.io/segmentation.html

Smith, F. 2018. *What Is a Service Mesh?* Entry on NGINX's tech blog. Accessed on 12 April 2019. Retrieved from https://www.nginx.com/blog/what-is-a-service-mesh/

Streams Quickstart Guide. N.d. Akka documentation page. Accessed on 14 April 2019. Retrieved from https://doc.akka.io/docs/akka/current/stream/stream-quickstart.html

The Scala Programming Language. N.d. Scala programming language's home page. Accessed on 10 April 2019. Retrieved from https://www.scala-lang.org/

Tour of Scala. N.d. Page on Scala language's documentation site. Accessed on 10 April 2019. Retrieved from https://docs.scala-lang.org/tour/tour-of-scala.html

Task graph. N.d. Sbt documentation page. Accessed on 14 April 2019. Retrieved from https://www.scala-sbt.org/1.x/docs/Task-Graph.html

Vagrantfile. N.d. Vagrant documentation. Accessed on 14 April 2019. Retrieved from https://www.vagrantup.com/docs/vagrantfile/

Warski, A. 2019. *Macwire*. Index of a GiHub repository. Accessed on 14 April 2019. Retrieved from https://github.com/adamw/macwire

What is Client-Server Architecture? N.d. Page on w3schools.in. Accessed on 10 April 2019. Retrieved from https://www.w3schools.in/what-is-client-server-architecture/

Xu, J. 2018. *Service discovery in microservice architecture*. Article on Hacker Noon. Accessed on 11 April 2019. Retrieved from https://hackernoon.com/understand-service-discovery-in-microservice-c323e78f47fd

Appendices

Appendix 1. Proof of Concept Git repository's readme file

Repository for thesis proof of concept

This Git repository contains the source codes for a proof of concept (PoC) environment as a part of
Juuso Minkkilä's (student id K1756) thesis *Health checks and service discovery in microservice environment*.

This repository's address is https://gitlab.labranet.jamk.fi/K1756/juuson-oppari and as of now requires JAMK LabraNet authentication and an invitation.

Upon release this repository will be partially copied over to a public repository at https://gitlab.com/SlightHeadache/jamk-thesis-public-release

The important folders from the point of view of the PoC are `Vagrant` and `Prototypes`. Please refer to the thesis itself or the repositorio's setup_notes.md when deploying the PoC.